CodeWarrior®

The PowerPlant Book

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/05 cbc

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

Foreword 13

Foreword to the First Edition . . . 14

Acknowledgments 15

Acknowledgments for the First Edition . . . 16

1 Introduction 19

Metrowerks Year 2000 Compliance . . . 20
What's New in This Release . . . 20
What You Should Know . . . 22
The Master Plan . . . 23
Background . . . 24
Basic Building Blocks . . . 24
Writing PowerPlant Code . . . 24
Starting Points . . . 26
Strategies For Learning . . . 26
Other Resources . . . 27
PowerPlant Information . . . 27
Object-Oriented Programming Information . . . 28
Third Party Books on PowerPlant . . . 29
Online Resources . . . 29
Conventions Used in This Book . . . 30
Your First PowerPlant Application . . . 30
Building the Interface . . . 31
Writing PPEdit . . . 45
What Next? . . . 51
Starting Your Own PowerPlant Projects . . . 52

2 Installing PowerPlant 57

PowerPlant Requirements . . . 57
Development Requirements . . . 57
Runtime Requirements . . . 58
Installing PowerPlant . . . 58
PowerPlant Source Code . . . 59

PowerPlant Documentation . . . 59
PowerPlant Example Code . . . 60
Installing Resource Templates . . . 61
Resorcerer . . . 62
ResEdit . . . 62
Rez . . . 62
Summary . . . 63

3 PowerPlant Conventions 65
Class and File Names . . . 65
Variable and Parameter Names . . . 66
Data Types . . . 67
Other Names . . . 69
Calling Macintosh Toolbox Routines . . . 70
Summary . . . 70

4 Application Frameworks 73
Reusable Code . . . 73
Procedural Code Libraries . . . 74
Class Libraries . . . 75
Frameworks . . . 75
Application Frameworks . . . 76
Framework Design Patterns . . . 78
Applications . . . 80
Event Handling . . . 80
Command Hierarchy . . . 80
Visual Hierarchy . . . 83
Messaging Systems . . . 85
Persistence . . . 87
Utilities . . . 88
Summary . . . 89

5 PowerPlant Architecture 91
Design Principles . . . 92
Multiple Inheritance . . . 92
Factored Design . . . 97

Factored Classes 99
Factored Behavior 101
Framework Implementation 103
Application Classes 104
Event Classes 105
Commander Classes 105
Visual Classes 108
Messaging Classes 110
Persistence Classes 111
Utility Classes 112
Basic PowerPlant Resources 113
PPob Resource 113
Mcmd Resource 115
RidL Resource 116
Txtr Resource 117
PowerPlant Development 119
Layout 120
Coding 121
Testing 122
Summary 123

6 Panes **125**

What Is a Pane 125
Pane Characteristics 127
Characteristics of Simple Panes 128
Characteristics of All Panes 128
Working With Panes 136
Creating a Pane 136
Drawing a Pane 142
Managing Pane Characteristics 145
Some Specific Panes 150
Summary 155
Code Exercise 156
Learning Paths 156
Basic Assumptions 158

The Interface. 159
Implementing a Custom Pane 163

7 Views **169**

What Is a View 169
View Characteristics 171
Subpanes 172
Image. 172
Scrolling 173
Coordinate Systems 174
Working With Views 180
Creating a View 180
Drawing a View 187
Managing Subpanes and the Visual Hierarchy 187
Managing the View Image. 189
Managing Scrolling. 190
Managing Coordinate Transformations 192
Some Specific Views 194
Summary 200
Code Exercise 200
The Interface. 201
Implementing a Custom View 206

8 Controls and Messaging **215**

What Is a Control 215
Control Characteristics 217
Control Values. 218
Control Descriptor 219
The Hot Spot 219
Broadcasting and Listening 220
Working With Controls 221
Creating a Control 221
Drawing a Control 225
Managing Control Characteristics 225
Broadcasting. 227
Being a Good Listener 231

Specific Control Classes 233
Summary 246
Code Exercise 247
The Interface. 247
CColorControl 255
The Controls Application 261
Intermission 268

9 Applications and Events 271
The Application Object 271
Application Class Hierarchy 272
Application State 273
Deriving an Application. 274
Initializing an Application 275
Set Debugging Options 276
Initialize the Heap 282
Initialize the Toolbox 283
Setup Memory Management 283
Check the Environment 287
Register PowerPlant Classes 288
Run the Application 290
Event Handling and Dispatch 290
PowerPlant and Apple Events 291
Summary 292
Code Exercise 293
The Interface. 293
Setting Up an Application 295

10 Commanders and Menus 307
Introduction to Commands 307
Command Chain 309
Target Handling 310
Duty Handling. 312
Command and Keystroke Handling 314
Making and Managing Menus 316
Menu Strategy 316

Menu-Related Resources 317
Command Numbers 318
Adding Menus 322
Responding to Menu Commands 323
When To Update Menus 325
Updating Menu Items 327
Working With LMenuBar and LMenu 331
Summary 333
Code Exercise 333
The Menu Resources 334
Implementing Menus 336

11 Windows 351

What is a Window 351
Window Characteristics 353
Window Attributes 353
Window Size and Zooming 358
Window Descriptor 360
Window Kind 360
Working With Windows 361
Creating a Window 362
Drawing a Window and Its Contents 368
Managing Window Behavior 370
Window Utilities in PowerPlant 374
Dealing with the Window Manager 377
Summary 378
Code Exercise 379
The Interface 379
The Windows Application 382

12 Dialogs 397

What Is a Dialog 397
Traditional Dialogs 397
PowerPlant Dialogs 398
LDialogBox Hierarchy 399
Dialog Characteristics 400

Working With Dialogs 401
Creating a Dialog. 402
Messages in Dialogs 406
StDialogHandler 408
Simple Movable Modal Dialogs 410
Traditional Dialogs 410
Summary 411
Code Exercise 412
The Simple Dialog Interfaces. 412
Implementing Simple Dialogs 413
The Complex Dialog Interface 420
Implementing a Complex Dialog 421

13 File I/O **433**

The Document Strategy 434
LDocApplication 436
What Is a Document 439
LDocument 440
LSingleDoc 443
What Is a File 444
What Is a Stream 446
LStream. 447
LFileStream 449
Saving and Opening Files 450
Implement an Application. 450
Implement a Document 451
Implement a Preferences File 454
Summary 455
Code Exercise 455
The Interface. 456
Implementing Documents 456

14 Printing **471**

Printing Strategy 471
LPrintout 474
LPrintout Characteristics 474

LPrintout Behaviors 476
LPlaceHolder 477
LPlaceHolder Features 478
LPlaceHolder Behaviors 479
Printing in Views and Panes 480
The Mac OS, LPrintout, and LPlaceHolder 482
Printing in PowerPlant 483
Building a Printing Hierarchy 483
Printing a Document 487
The Print Record 492
Printing Utilities 493
Summary 494
Code Exercise 495
The Interface 495
Implementing Printing 499

15 Periodicals and Attachments 505
Periodicals 506
What Is a Periodical 506
Periodical Characteristics 507
Working With Periodicals 507
Attachments 512
What Is an Attachment 512
Attachment Strategy 514
Attachment Characteristics 515
Working With Attachments 518
Specific PowerPlant Attachments 522
Summary 524
Code Exercise 525
The Interface 526
Implementing Goodies 528
Looking Backward, Looking Forward 539

A PowerPlant Utilities 541
PowerPlant Utilities Overview 541
Classes Discussed Elsewhere 542

More Utility Classes 542
LClipboard 543
Arrays 547
LString 558
LSharable 559
UScreenPort 560
UDrawingState 560
UDrawingUtils 562
UKeyFilters 565
UProfiler 567
UReanimator 568
UResourceManager 568
UTextTraits 569

B Resource Notes 571
PowerPlant-Specific Resources 571
Standard Resources 572
PP Copy & Customize.ppob 572
PP Copy & Customize.rsrc 573
PP Action Strings.rsrc 575
PP DebugAlerts.rsrc 575
PP Document Alerts.rsrc 575
PP AppleEvents.rsrc 576
ColorAlertIcons.rsrc 576
ToolServer and Rez 577
Using ToolServer 577
Rez 577
DeRez 579

Index 581

Foreword

I started developing PowerPlant a little over five years ago. All along, PowerPlant has adapted to changes and new features in both the Mac OS and the C++ language. Managing the growth and increasing complexity of the framework is my greatest challenge.

PowerPlant began as a System 7 framework, with support for event handling, menus, files, windows, controls, and text editing. Over the years, it's expanded to support new Mac OS features such as the Drag-and-Drop, Open Transport networking, and the Appearance Manager.

Concurrently, the CodeWarrior compiler kept implementing additional C++ language features. I gradually began using template classes and runtime type identification. Despite having a syntax heavily laden with angle brackets that confuses new C++ programmers, I think they've made the code more robust and flexible.

In my foreword to the first edition of this book, I mentioned that PowerPlant encourages a "buffet" style of programming where you can pick out the classes that interest you. Although the buffet has many more selections now, Carl Constantine keeps everything in order in this second edition of the book. Carl's rich descriptions will entice you to dig in and sample what PowerPlant has to offer.

Bon appetite!

Gregory Dow

Foreword to the First Edition

I started to write an application framework just a few months after buying my first Macintosh. I thought it would be a good way to learn Mac programming and that I could use the framework as the basis for the great software I would someday write.

Well, I couldn't come up with an idea for an indispensable utility program or an addictive game, so I just kept working on frameworks. I've written more than a dozen over the years. Some are just partial implementations to prove a concept. Others I used to write small programs. A couple, including PowerPlant of course, have been published.

With PowerPlant, I've tried to build a tool that enables you to create your own great software. In fact, PowerPlant is really a set of tools, because it contains families of classes that you can use together or separately. It's not an "all or nothing" framework. You can just use portions of PowerPlant, and can easily integrate specific classes into existing programs.

This design encourages what I call a "buffet" style of programming. You look over all the classes, pick the ones that look interesting, and then go back for more later. Furthermore, the wide variety of classes tempts you to experiment with features that you might not have tried on your own.

In this book, Jim Trudeau piles his plate high with PowerPlant's offerings and vividly describes the subtle flavors. Jim clearly explains not only the details but also the philosophy of programming with PowerPlant. I'm thrilled that he wrote this book because I know it will spur you to use PowerPlant to write great software.

Gregory Dow

Acknowledgments

This revised manual has been a joy to work on for me. The feedback Metrowerks has received since the first version has been excellent and we hope to see more. I want to thank all of you for your feedback and suggestions.

With any change of authors, there's always some trepidation as you try to follow in someone else's well laid footsteps. I would like to thank Jim Trudeau, my mentor since I joined Metrowerks, for his guidance, advice, and constructive criticism of my work. Jim offered many suggestions for parts of this book and I've endeavored to implement them to make this manual clearer in some areas.

Richard Atwell provided many suggestions and helped clear up resource and stationery issues.

John Daub provided a lot of feedback on class registration and debugging issues. John was also extremely influential in helping me understand some of the intricacies of PowerPlant and C++. Thanks John!

Greg Dow, PowerPlant's chief architect, has kept PowerPlant up-to-date. He and other engineers have guided PowerPlant's evolution, added new functionality, streamlined code, and much more. Greg has made PowerPlant both a joy and easy to use.

Last but not least, I want to thank the folks at "the MOST Online," particularly radar pangaean (no capitals intentional) for reviewing an early version of this rewrite and allowing me to use them as guinea pigs.

Carl B. Constantine

Acknowledgments for the First Edition

This manual has been an extraordinary project, both for me and for Metrowerks. The credits at the back of this manual do not do justice to the many fine people who had a hand in it.

I especially want to thank Greg Galanos, President of Metrowerks. He recognized that there was a need for a new approach to PowerPlant. True to the company motto, he listened and he acted.

Greg Dow, PowerPlant's parent and chief designer, deserves special mention. He and the other fine engineers at Metrowerks have created a fantastic application framework. As you'll see when you delve into the inner workings of PowerPlant, this is a world-class piece of code. It is elegantly designed and beautifully realized. Greg's early design decisions made my job—explaining PowerPlant to you—easier, more fun, and more rewarding. I can promise you that his design will make your learning experience far more enjoyable as well. Best of all, his efforts in PowerPlant will allow you to write better, more reliable, and more powerful software.

Phil Bornstein and Avi Rappoport blazed a trail into the unknown by writing the first PowerPlant manual. Thanks!

Harold Ekstrom deserves praise both as an outstanding tech editor and code writer. Harold wrote most of the code in the exercises in this book. He is an experienced PowerPlant master, and another of the fine people who put this book together for you.

Eric Scouten, the Constructor constructor at Metrowerks, deserves thanks for answering those technical questions I came up with at odd hours. There is a lot of his expertise in these pages.

Last, but not least, there is the PowerPlant Dream Team.

I started writing this book on October 2, 1995. By one of those curious twists of synchronicity that the universe occasionally throws at us, Steve Jovanovic posted a note on `comp.sys.mac.programmer.codewarrior` on that very day. He asked if anyone might be interested in joining him in an adventure—learning all about PowerPlant.

His note said, in part, "If you're interested in a close-knit, small group, with fast-paced results, in the tradition of the Macintosh Warrior zeal, let's do it!" I wrote to Steve and expressed an interest in testing this documentation out with the group. We thought we'd have 10 or 15 people—30 at the most. Neither of us had any idea what we had let ourselves in for.

Within two weeks there were more than 200 people in the self-styled Dream Team. It had a web site, several mail lists on different topics, and small teams of programmers all over the world exploring PowerPlant and sharing what they learned. They became my official beta testers. Every week I'd ship a couple of draft chapters to guide their exploration. Then I'd wait for the feedback. Boy, did I get feedback! They really took the word "zeal" to heart.

These people made hundreds of suggestions for improvements, both large and small. Sometimes it was just a misplaced comma. Other times, a casual suggestion would send me off for two hours of reorganizing and rewriting. They found a few flaws—happily not many. By far, most of the comments were not for mistakes in the manual, but for ways in which it could be better. Almost every one of those suggestions was applied to this book.

So, to the PowerPlant Dream Team—especially Steve Jovanovic who had the idea and Alex Tang for setting up the network sites—a tip of the author's hat. And a special thank you to those who were most prolific and consistent in providing invaluable, real-world user feedback: Bob Tucci, Richard Atwell, Alex Bangs, Carl Constantine, Jim Jarrett, Jim Van Peursem, and Peter Zeigler. These folks were my most prolific editors, but they were not alone. I had suggestions from dozens of people on four continents!

What you have in front of you, then, is better than it otherwise would have been. To the Dream Team, and to everyone else who had a hand in this book inside and outside of Metrowerks, I say a heartfelt thank you.

This book really is the product of the "Ahnold" ethic. I hope you have as much fun learning from it as I had writing it.

Jim Trudeau

1 Introduction

Welcome to PowerPlant®!

Although some folks like to skip introductions, we strongly recommend you read this one. Why? It serves several important purposes, not the least of which is that you write a complete PowerPlant application! This chapter is really your introduction to PowerPlant, not just this book.

What is PowerPlant? PowerPlant is Metrowerks' world-class, object-oriented, Macintosh application framework. Using PowerPlant you can quickly build the reliable interface your program needs to be successful with your customers or clients. PowerPlant gives you a complete, powerful, ready-to-run application that takes advantage of the latest features of the Mac OS. All you have to do is add the content that makes your application unique.

Exactly what PowerPlant is—what its parts are, and how to use them—is what this book is all about. This book gives you a thorough understanding of the design principles and implementation details behind PowerPlant. Along the way you'll discover that PowerPlant is not that hard to learn. It is complex, sure. But it is also logical, organized, and very well designed.

This Introduction gives you a road map so that you will have an idea of what to expect as you read this book. At the end of the introduction you write your first PowerPlant application. The Introduction discusses these principle topics:

- What's New in This Release—overview of changes to PowerPlant and this manual.
- What You Should Know—the background knowledge this book assumes you have.
- The Master Plan—the structure of the book, its chapters, and how they are organized.

- Starting Points—where you should jump into the book if you are already conversant with some of the ideas and concepts.
- Strategies For Learning—how to use the contents of this book to enhance your learning experience.
- Other Resources—where to go to learn more about frameworks, object-oriented design, and PowerPlant.
- Your First PowerPlant Application—in which you write a simple text-editor application from scratch, using PowerPlant.

These sections will help you find your way through what will be an enjoyable and worthwhile project: learning how to use PowerPlant.

Metrowerks Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: `http://www.metrowerks.com/about/y2k.html`.

What's New in This Release

PowerPlant is a growing, evolving framework. There have been many changes and additions to PowerPlant. Some of the more significant changes include:

- new registration macros

- updated existing classes
- namespace support
- support for Apple's Appearance Manager and Navigation Services

Registration macros

Previous versions of PowerPlant had a difficult class registration mechanism. Registering PPob-based classes is now one simple command, `RegisterClass_(ClassName)`. This applies to classes within PowerPlant as well as any custom PPob class you derive.

The `RegisterAllPPClasses()` method was used to register many of the common PowerPlant classes in one operation. However, this added a lot of unneeded code to your final binary. This method is still supported for compatibility purposes, but you should update your code to use the new `RegisterClass_()` macro and only register the classes you *need*.

See also "Register PowerPlant Classes" on page 288.

Updated classes

Many PowerPlant classes have been improved, expanded and debugged. Some changes include:

- New LString functions for dealing with floating point numbers using flexible formatting options.
- New LString functions for using Int16 numbers
- New LStringRef class which lets you use LString functions to manipulate an arbitrary string
- New LTabGroupView class. All Panes that are also Commanders enclosed within a LTabGroupView are part of a tab group. Similar to LRadioGroupView.
- New UModalAlerts class (within `UModalDialogs.h` & `UModalDialogs.cp`) with wrappers for the Toolbox `Alert()` calls. The wrappers deactivate the desktop and supply an event filter for handling events while the alert window is displayed.

Namespace support

CodeWarrior now supports the use of C++ namespaces. PowerPlant has its own namespace: `PowerPlant`. This book does not cover the use of namespaces. Namespaces are turned off by default.

See also PowerPlant Release notes for more detailed information.

Updated system support

PowerPlant includes classes to support Apple's Appearance Manager and Navigation Services. These classes are not yet covered in this book.

See also PowerPlant Release notes for more information on how to use these classes.

What You Should Know

This book assumes you are familiar with:

- Macintosh application programming
- the C++ language
- the CodeWarrior development environment

To be familiar with Macintosh application programming means you understand concepts such as the event loop, creating and managing menus, creating and managing windows, creating and managing dialogs and control items, idle time processing, memory management, and so forth.

To be familiar with C++ implies that you not only understand C++ syntax (such as what a class is, a constructor, a destructor, and so forth), but also that you are familiar with the features of an object-oriented language such as inheritance, polymorphism, overriding, and overloading.

If you are unfamiliar with these topics, you should pursue them before you expect to use PowerPlant effectively. The extensive documentation available with CodeWarrior can help.

The CodeWarrior Reference CD contains electronic versions of several programming books including: *Learn C on the Macintosh* by Dave Mark; *Programming Starter Kit for Macintosh* by Jim Trudeau; and *Learn C++ on the Macintosh* by Dave Mark. These books cover everything you need to know about programming languages as well as Mac programming, from memory management to drag and drop using an object-oriented approach in C. The exercises use the CodeWarrior environment.

These and other third-party books on Macintosh programming and C++ are available at computer bookstores everywhere, and directly from Metrowerks at the CodeWarrior Store.

For more information about the CodeWarrior development environment, consult the CodeWarrior documentation, especially the *IDE User Guide* and the *Debugger User Guide*. These will familiarize you with the environment.

In the code exercises in this book you will regularly encounter and use PowerPlant's view-editing sidekick, Constructor. This book explains what Constructor is and what you use it for. However, we will not go into detail about how to *use* Constructor. While performing the code exercises in this book, if you have questions about Constructor, you can consult the *Constructor for PowerPlant Guide* that comes with the CodeWarrior documentation.

See Also "Other Resources" on page 27 for more on PowerPlant and object-oriented programming.

The Master Plan

This book is divided into a series of chapters. The chapters can be grouped into three sections:

- Background
- Basic Building Blocks
- Writing PowerPlant Code

Each section has several chapters.

Background

Chapters 2 through 5 make up the Background section. Chapter 2, "Installing PowerPlant" discusses installing PowerPlant, and PowerPlant resource templates. Chapter 3, "PowerPlant Conventions" discusses name and style conventions in PowerPlant code. Chapter 4, "Application Frameworks" discusses the nature of application frameworks in general, and the kinds of design features you are likely to encounter in an application framework. Chapter 5, "PowerPlant Architecture" introduces you to the basic design of the PowerPlant framework, and the classes, objects, and resources you will encounter as you use PowerPlant.

The chapters in the Background section are informational. They give you the big picture overview of PowerPlant, and how PowerPlant fits into the world of object-oriented programming. These chapters do not have any code exercises.

Basic Building Blocks

Chapters 6 through 8 comprise the Basic Building Blocks section. The chapters in this section give you an early look at PowerPlant's most important objects. The chapters are:

- Chapter 6, "Panes"—all about panes, the fundamental visible objects in PowerPlant.
- Chapter 7, "Views"—pane containers that create a visual hierarchy.
- Chapter 8, "Controls and Messaging"—more about panes, and how they communicate with each other.

In these chapters you write real PowerPlant code. You will use this experience in the Writing PowerPlant Code section.

Writing PowerPlant Code

Chapters 9 through 15 cover various aspects of PowerPlant application programming from a task-based perspective. The chapters are:

- Chapter 9, "Applications and Events"—an introduction to the command hierarchy.

- Chapter 10, "Commanders and Menus"—adding menus to a PowerPlant application.
- Chapter 11, "Windows"—creating and managing windows, including floating palettes.
- Chapter 12, "Dialogs"—creating and managing dialogs, including modal, movable modal, and modeless dialogs.
- Chapter 13, "File I/O"—saving and opening files.
- Chapter 14, "Printing"—printing a document with PowerPlant.
- Chapter 15, "Periodicals and Attachments"—idle time processing and attachments.

Chapters 6 through 15—the Basic Building Blocks and Writing Code sections—are more practical and detailed. Here you find the solid information you need to answer the real-world question, "How do I use PowerPlant?" Each chapter has two parts.

The first part of the chapter discusses PowerPlant fundamentals. It introduces you to the classes involved in that chapter and how you use them. We'll discuss their member functions and data members, their inheritance chain, and the common situations in which you use the class.

The second part of each chapter is a code exercise. In this part of the chapter you write real PowerPlant code following step-by-step instructions. This gives you an opportunity for hands-on practice with the real thing.

The code exercises are all application-based. PowerPlant is first and foremost an *application* framework, after all. However, you can use PowerPlant classes for other programming projects such as code resources and shared libraries. As you learn about PowerPlant, you'll see how the very design of this framework allows you to use just those parts of the framework that you need for your particular project.

A good application framework has lots of utility functions to aid you in your work. PowerPlant is no exception. We discuss many task-based utilities in the chapters as we work through the book.

However, you should consult Appendix A, "PowerPlant Utilities" for a potpourri of additional helpful ditties in PowerPlant.

Starting Points

The chapters in this book are more or less sequential. Each chapter builds on the knowledge gained in previous chapters. As a result, the typical path through these sections is linear. For the most part, you start at the beginning and work through to the end.

Everyone who is new to PowerPlant should read Chapter 2, "Installing PowerPlant" to ensure that you are familiar with the various source and header files you will encounter. You should also read Chapter 3, "PowerPlant Conventions" so you understand the naming conventions used throughout PowerPlant.

Programmers unfamiliar with application frameworks in general should continue with Chapter 4, "Application Frameworks" to learn about the common design patterns you encounter. If you are an experienced object-oriented programmer and you know about application frameworks and the design patterns you encounter in them, you can safely skip this chapter. From that point on, if you are new to PowerPlant you should read all the remaining chapters sequentially.

Experienced or somewhat experienced PowerPlant programmers can examine the Table of Contents for a chapter that addresses an application programming topic of interest, and go to that chapter. You can always revisit a topic in some other chapter if you encounter something you don't understand.

Strategies For Learning

The best way to learn PowerPlant is to learn about the concepts, then play with example code. This simple fact is reflected in the structure of the code-intensive chapters: we discuss fundamentals first, and then work with real code right away.

You can skip the code exercises if you wish. If you do, the lessons learned in the fundamental section of each chapter will remain theoretical. While theory is both necessary and useful, theory can only take you so far when it comes time to implement real working solutions in code. We recommend that you also work with the code exercises in each chapter to master the intricacies of PowerPlant.

As you perform the code exercises, think about what you've learned in the fundamentals discussion. Apply those lessons to the problems presented in the coding steps. All of the code you need will be included in with the steps, tutorial fashion. However, feel free to attempt your own solution to the problem.

Solving problems in your own way is the best way to learn. You may discover on some occasion that your solution fails because you didn't understand some subtlety in the framework design or the code. That's fine. The lessons you learn from the failure will be far more valuable and stick with you much longer than those you learn from blindly copying code in the tutorial steps.

As you become familiar with the general structure of the PowerPlant framework, you'll gain confidence. You'll find your own solutions becoming more reliable, and more accurate. Very soon you'll be able to apply these lessons to your own programs. When you do, you'll find that you can create better, more powerful, and more reliable Macintosh software faster than you ever could before.

Other Resources

After you have completed this book, or those parts of the book you find of personal interest, you may want to go further. This book does not cover a variety of advanced PowerPlant programming topics. However, do not be alarmed. There are places to go to learn more about PowerPlant.

PowerPlant Information

Your first and most common stop on the path to learning more about PowerPlant will be *PowerPlant Revealed*. This document is

available in electronic form as part of the CodeWarrior documentation. It is a series of HTML files describing all the PowerPlant classes, their data members and member functions. This hypertext reference allows you to locate specific functions, trace class derivations, look up inherited data members, and so on.

The comments found in the PowerPlant source files are another valuable source of insight about PowerPlant. This information is frequently duplicated in the *PowerPlant Revealed* HTML Reference.

Metrowerks provides additional PowerPlant documentation as well. The *PowerPlant Advanced Topics* manual covers a variety of subjects in-depth. The precise nature of what's available varies from release to release, and as PowerPlant evolves. Check out the PowerPlant documentation folder for the latest information.

PowerPlant is a hot topic on various on-line services and the Internet newsgroups `comp.sys.mac.oop.powerplant` and `comp.sys.mac.programmer.codewarrior`. Experienced PowerPlant programmers and Metrowerks employees often answer questions in these groups. The newsgroup discussions are archived on the Reference CD, in the Metrowerks Documentation folder. You can search through the archives for information on a wide variety of PowerPlant topics. They are a useful resource.

Metrowerks maintains a World Wide Web site where you can find updates to PowerPlant classes as well as third-party classes based on PowerPlant. These classes extend PowerPlant's functionality in unique and useful ways. You can find the web page at:

```
http://www.metrowerks.com/db/powerplant.qry?function=form
```

Object-Oriented Programming Information

There is a wealth of knowledge available if you want to learn more about object-oriented programming in general. There are several books on the subject. A quick sampling of titles includes:

- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley, 1995).

- *Design Patterns for Object-Oriented Software Development* by Wolfgang Pree (Addison-Wesley, 1994).
- *Developing Object-Oriented Software for the Macintosh* by Neal Goldstein and Jeff Alger (Addison-Wesley, 1992).
- *C++ Primer Plus, Third Edition* by Stephen Prata (Waite Group Press, 1998)
- *Effective C++ , 2nd Ed*. by Scott Meyers (Addison-Wesley, 1997).
- *More Effective C++* by Scott Meyers (Addison-Wesley, 1996).

Third Party Books on PowerPlant

If you want to learn more about programming with PowerPlant, there are a few books to look at. They include:

- *The Metrowerks CodeWarrior Professional Book* by Dan Parks Sydow (Ventana, 1998)
- *CodeWarrior:Software Development Using PowerPlant* by Jan Harrington (AP Professional, 1996)
- *Metrowerks, CodeWarrior Programming* by Dan Parks Sydow (M&T Books, 1995)

Online Resources

In addition, there are many web sites dedicated to object-oriented programming. Some of these sites are specific to the Mac OS.

- `http://www.codewarrrior.org/`
- `http://www.apple.com/developer/`
- `http://www.themost.org/`
- `http://cafe.AmbrosiaSW.com/alt.sources.mac/macintosh-c/`
- `http://www.cernet.com/~mpcline/C++-FAQs-Lite/`
- `http://www.lysator.liu.se/c/index.html`

Conventions Used in This Book

This book uses various typographical conventions to make reading easier.

`Computer Voice` is used for source file names and code examples.

Bold text is used when refering to menu names and items, as well as specific buttons in dialogs. For example, click the **Save** button.

Key words or book names are presented in *italic* text.

Additionally, this book may reference a hypothetical application called HyperApp with main application files `CHyperApp.cp`, `HyperApp.rsrc`, and `HyperApp.PPob`. Unless otherwise stated, descriptions of this hypothetical application apply to all PowerPlant applications.

Your First PowerPlant Application

Assuming that you have never done any PowerPlant programming, you're in for a treat. Admittedly, you're not going to learn a lot of PowerPlant specifics by following the steps in this exercise, because you haven't got all the background information that will help you process and categorize the new knowledge.

Nevertheless, by writing a real application right away you will see first hand the tremendous advantages that PowerPlant gives you when you're developing software. And that should give you all the incentive you need to absorb the background material. Soon you'll be ready for some intense PowerPlant coding.

NOTE: This exercise assumes you have installed PowerPlant. If you have not installed PowerPlant and you wish guidance on what you need to do, Chapter 2, "Installing PowerPlant" will help. In addition, the screenshots in this book are from the latest version of Constructor. If you are using an earlier version, the appearance of the Constructor windows and some features may vary.

So, let's get coding!

We're going to do this in two stages. In the first stage, you use Constructor to build a visual interface for a very simple text-editing application. In the second stage, you'll write the code to create the application.

Building the Interface

1. **Open the project file.**

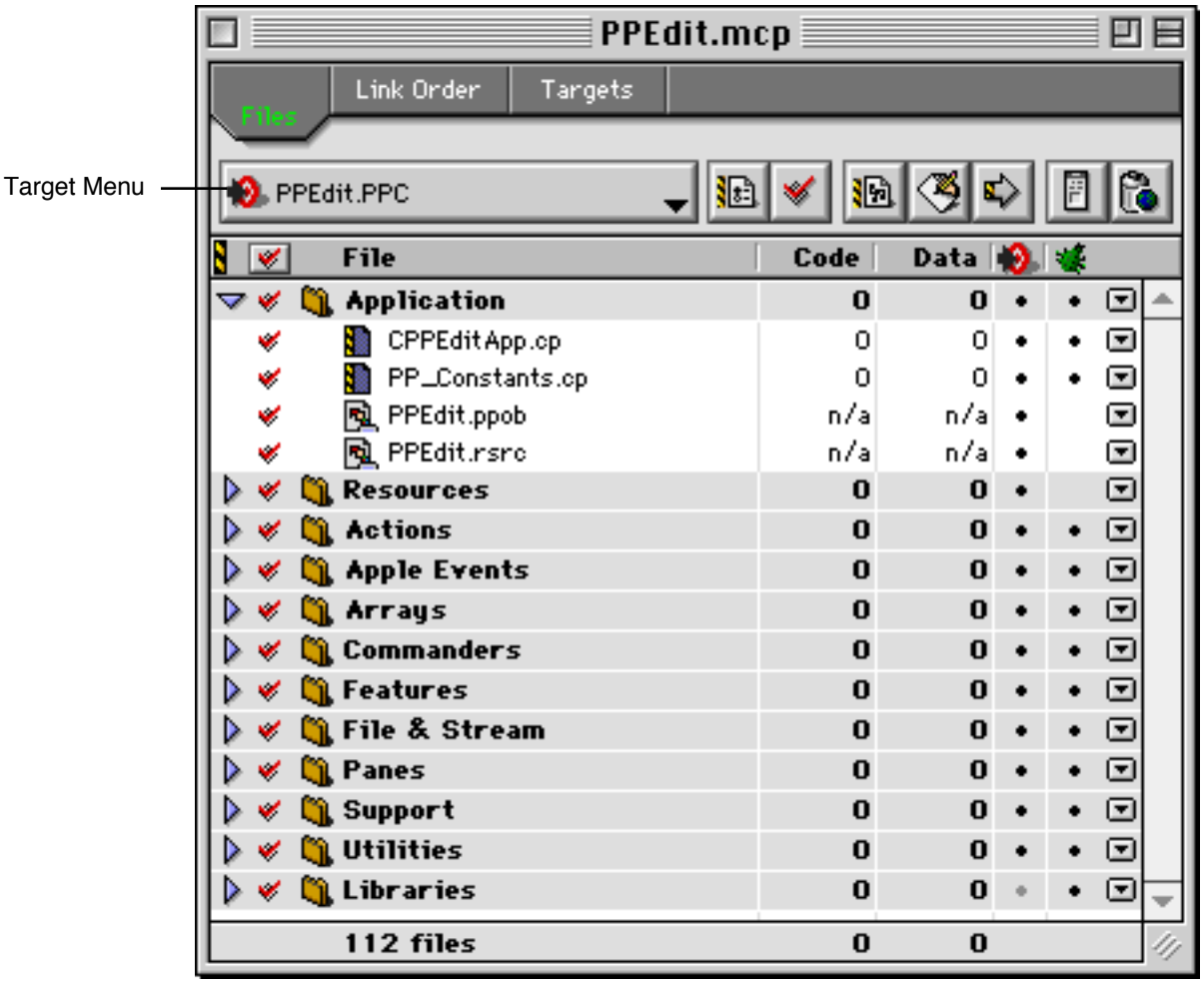
   The necessary files are in the "Chap 01 Start Code" folder, inside the "PP Book Code" folder. Locate this folder on the Reference CD-ROM and copy it to your hard drive.

   As always, you have two ways you can open a file. Double click the `PPEdit.mcp` project file icon, or launch CodeWarrior IDE and open the project using the **Open** command from the **File** menu.

   When you do, a project window opens. It should look like Figure 1.1. Choose the appropriate target for your computer (68K or PPC) from the Target Menu on the Project Window. The 68K target will work on all Mac OS computers.

   See also The *IDE User Guide* for more information on "Targets."

Figure 1.1 The PPEdit project



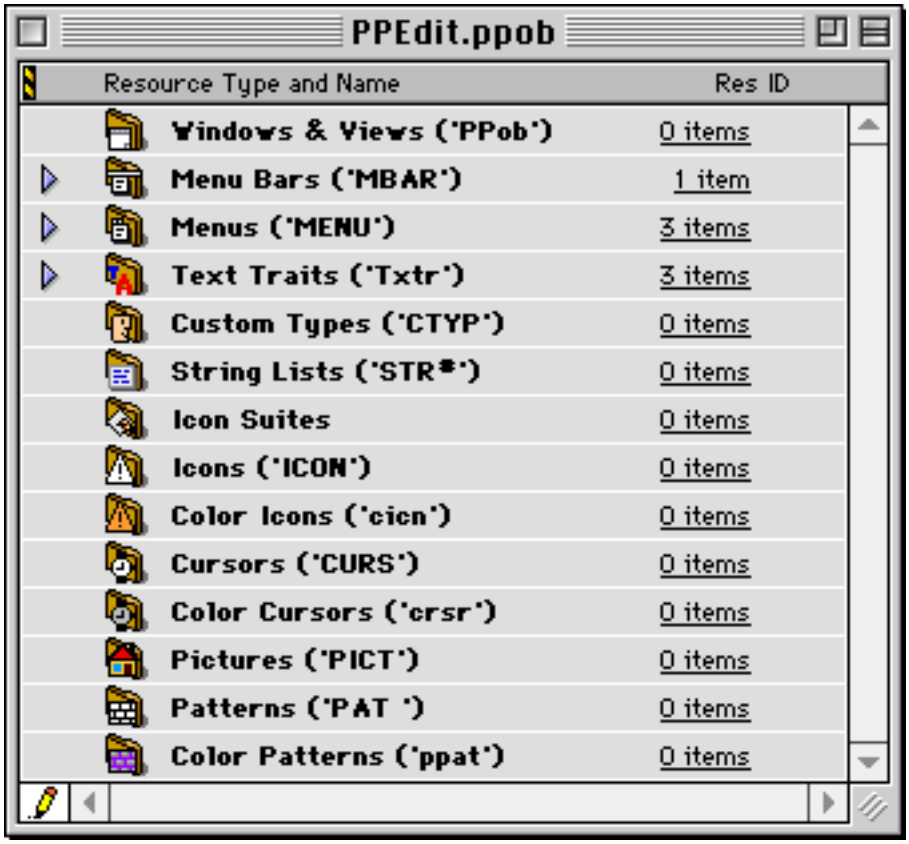
2. Open the Constructor file.

In this project we have divided the application resources into two resource files, `PPEdit.ppob` and `PPEdit.rsrc`. The file creator for the .ppob file is Constructor. The file creator for the .rsrc file is ResEdit. You will not have to modify the .rsrc file at all.

NOTE: Constructor is used to create and edit PPob, Txtr, MBAR, MENU, Mcmd, CTYP and bitmap resources. We'll discuss PowerPlant resources in detail in Chapter 5, "PowerPlant Architecture".

In the project window, double-click the `PPEdit.ppob` file. Constructor launches and opens a Constructor project window, as shown in Figure 1.2.

Figure 1.2 The PPEdit.ppob project window



There are already menu bar, menu, and text trait resources in this file. In this exercise you add a window and view (PPob) resource that describes the text editor's visual interface.

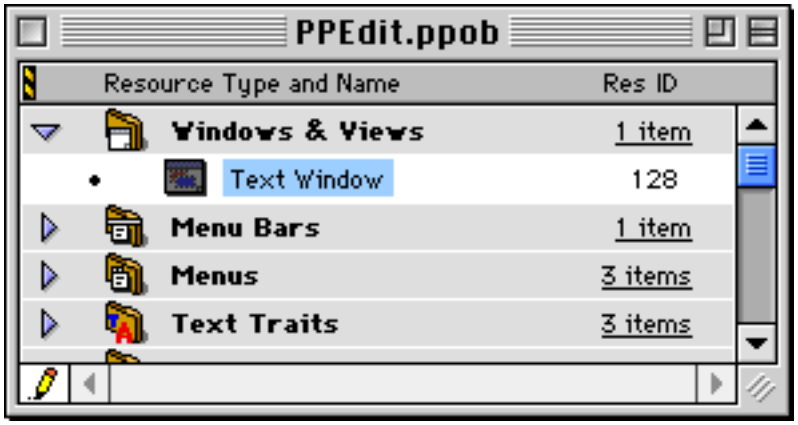
3. **Create a window.**

   Choose **New Resource** (command-K) from the **Edit** menu. A dialog appears, as shown in Figure 1.3. Set the values in this dialog to match the illustration.

Figure 1.3 Creating a new PPob resource

Set the resource type to Layout. Set the view type to LWindow. Enter the name of the new resource. Set the resource ID to 128. Then click the Create button. The new resource appears in the project window, as shown in Figure 1.4.

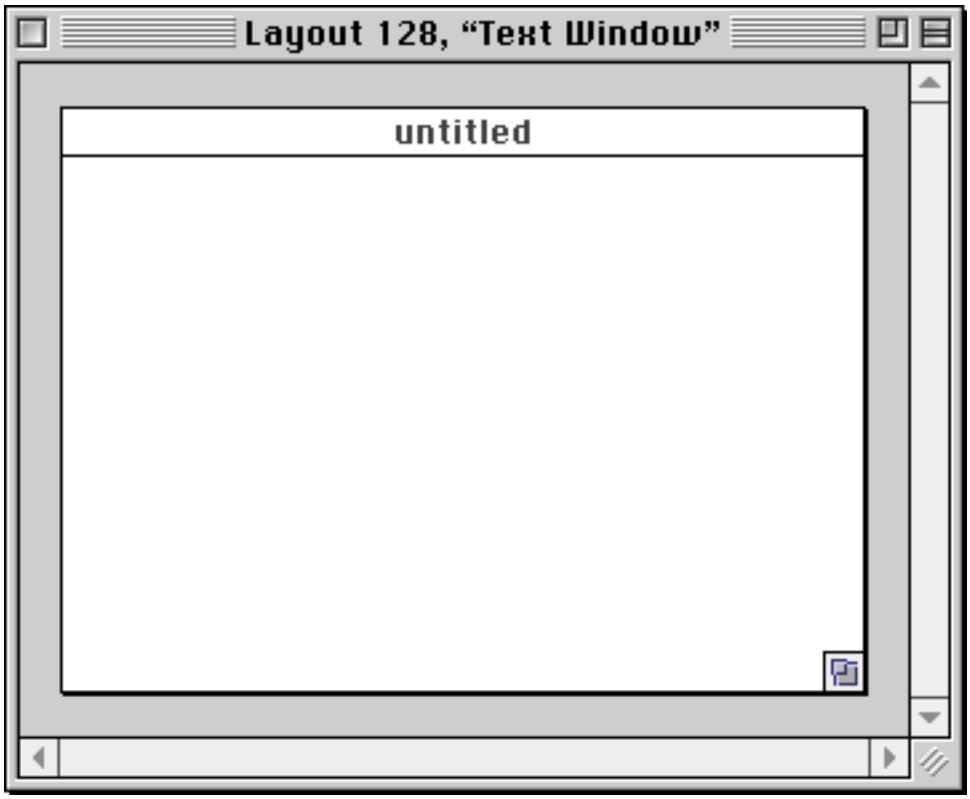
Figure 1.4 The new resource



You have just created the beginnings of a PPob resource. The ID number is significant. You use the ID number to tell PowerPlant which PPob resource to use when creating a new window. The name you assign isn't significant. The name helps you identify the correct resource when you have several. In this application you have one PPob resource. When you're through, the PPob resource will define the entire view and its contents.

In the Constructor project window, double-click the new Text Window resource. A window opens that displays the new (untitled and empty) window, as shown in Figure 1.5. This is the Constructor layout window. The layout window itself identifies that you are working with a PPob resource, with ID number 128, named "Text Window."

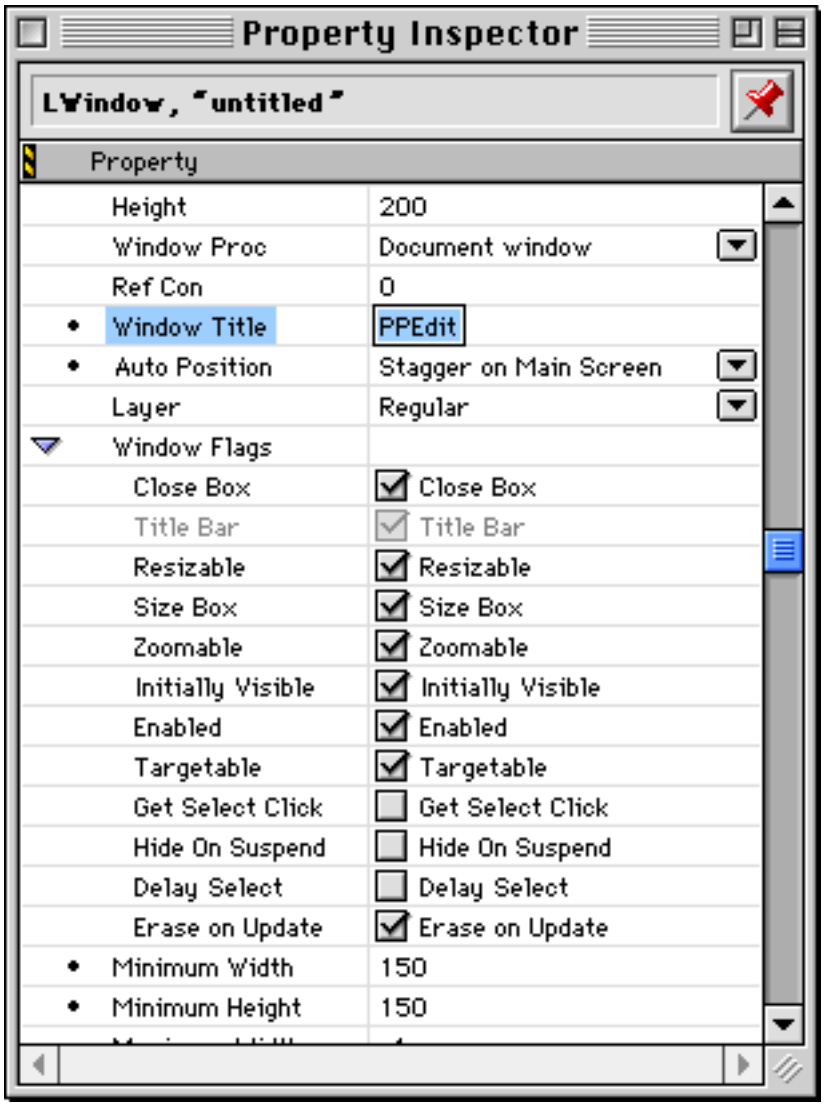
Figure 1.5 The new window



4. **Set the window characteristics.**

   You can double-click the area of the new window, or click it once to select it (four little selection handles will appear at the corners) and choose **Property Inspector** from the **Window** menu. When you do, the Property Inspector window appears, as shown in Figure 1.6.

Figure 1.6 The window properties



This window allows you to specify the various characteristics of the window, including location, size, type, title, and several other fea-

tures. Set up your new window to match the characteristics displayed in Figure 1.6.

To do this, you need to set the window title, set its minimum size to 150 by 150, and set its auto position to be **Stagger on the Main Screen**. All the other characteristics use default values.

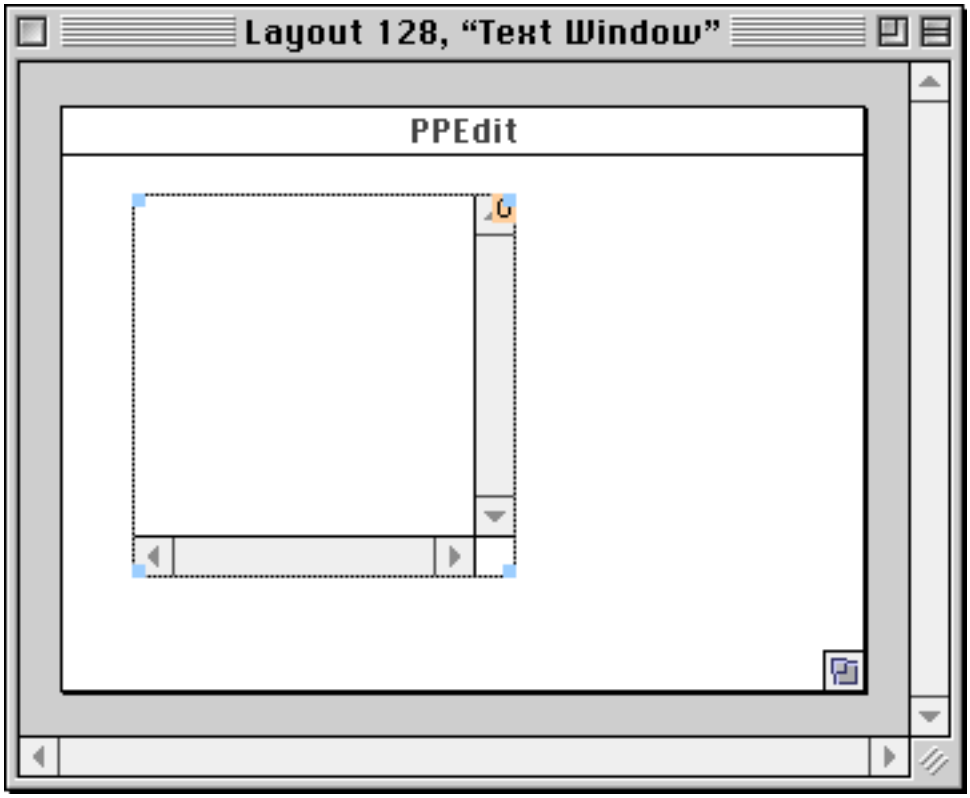
When you're through, close the property inspector window. Save your work. It's always wise to save your work often.

5. **Create a scrolling view.**

   In any text-editor, you want to be able to scroll text. In PowerPlant, that means you want a scrolling view. Adding a scrolling view to your new window is simple.

   First, make sure the Catalog window is visible. If it isn't, choose the **Catalog** item from the **Window** menu.

Figure 1.7 Adding a scroller



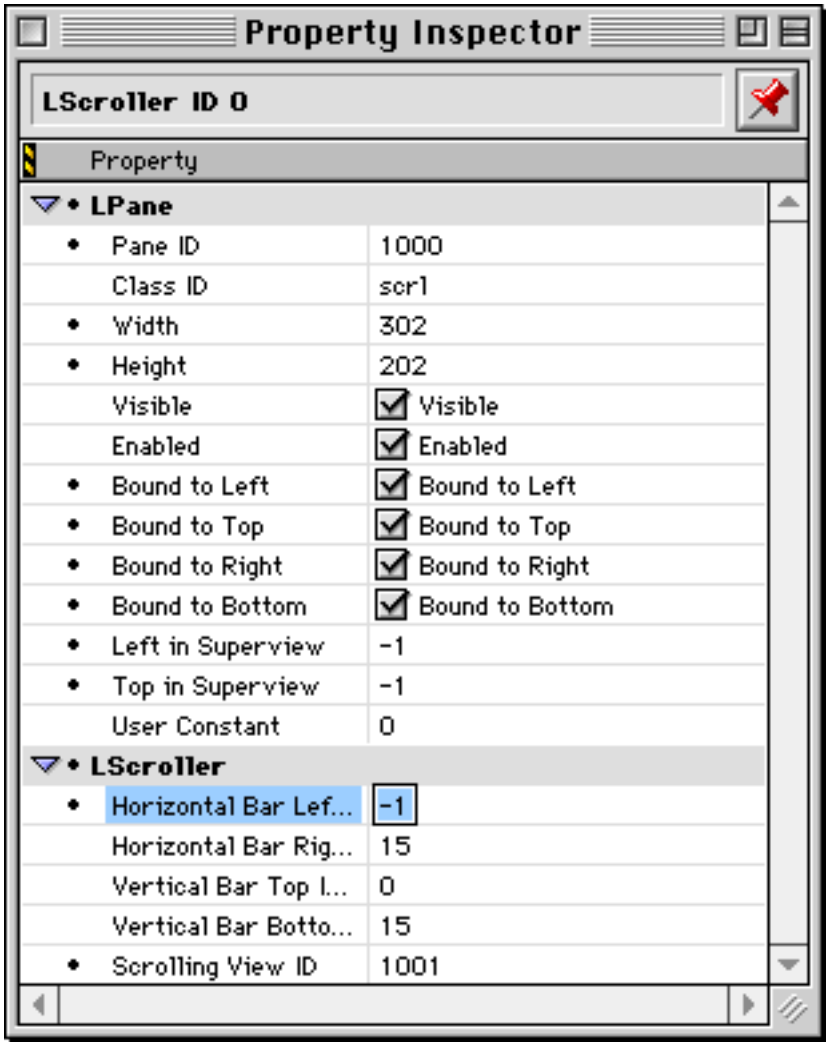
When the window is visible, display Views, then click and drag the LScroller icon from the Catalog window and drop it anywhere in the layout window, as shown in Figure 1.7.

6. **Set the scroller characteristics.**

   Double-click the new scroller, or select it to see its properties in the Property Inspector. Set the values to match those shown in Figure 1.8.

   Set the top left to (-1,-1), the width to 302, and the height to 202. These dimensions are one larger on all sides than the underlying window. Set the new view to be bound on all sides to the "Superview" by clicking the appropriate check boxes. Views and superviews are a hierarchy of drawing areas. In this case the superview is the window. Binding means that as the window resizes, the scrolling view (which is inside the window) stays pinned to the top left of the window and resizes along with the window.

Figure 1.8 Scroller characteristics



Set the Pane ID to 1000. Each pane in a view usually has a unique ID.

Set the Scrolling View ID to 1001. Putting that number in the Scrolling View ID tells PowerPlant that this scrollbar controls the pane numbered 1001. In the next step you'll create pane 1001, the text edit pane.

Finally, set the horizontal scroll bar left indent to -1 so that you won't have a horizontal scroll bar. The other values are defaults.

When you're through setting the scroller characteristics, close the Pane property inspector window and save your work.

TIP: Have you noticed a pattern here? You create the object you want, then you modify its characteristics. Some characteristics—like size and location—you modify interactively by dragging and resizing the object in the window. To set most characteristics, you open the Pane property inspector window.

---

7. **Create and set the characteristics for the text edit pane.**

   Now, click and drag the LTextEditView icon from the Catalog window, and drop it inside the scrolling view. Then set the properties for the new text edit pane.

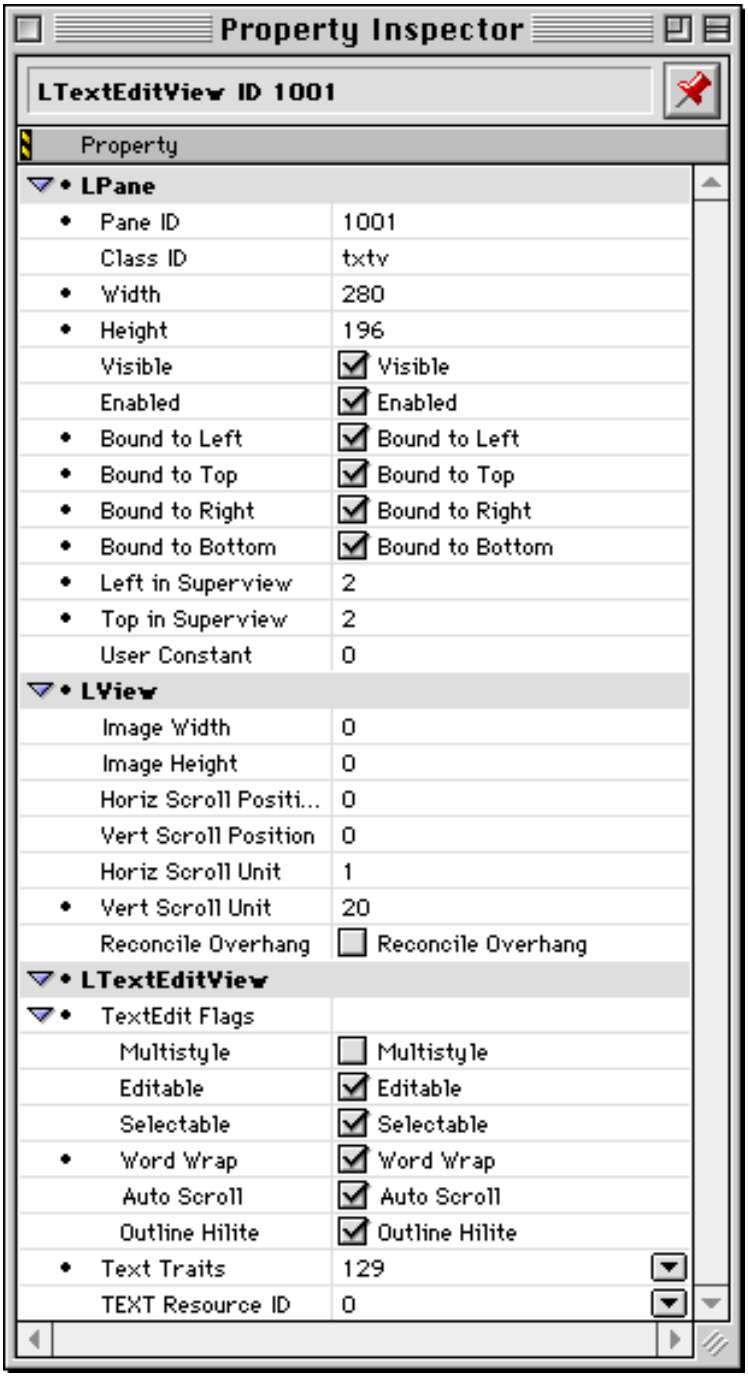
   Set the characteristics to match those in Figure 1.9.

   Set the top left to (2, 2), the width to 280, and the height to 196. These dimensions mean that the text edit pane is slightly smaller than the window (and allows room for the scroll bar on the right). As a result, the text in the pane will not come right up against the very edge of the window or the scroll bar.

   Set the edit text pane to be bound on all sides to the superview. The superview in this case will be the scroller pane. What this means is that as the window resizes, the text pane will resize along with the scroller and the window.

   Set the Pane ID to 1001. This matches what you set in the scroller characteristics. Set the text traits ID to 129. We'll look at text traits in step 9. Set the vertical scroll unit to 20.

Figure 1.9 Edit text characteristics



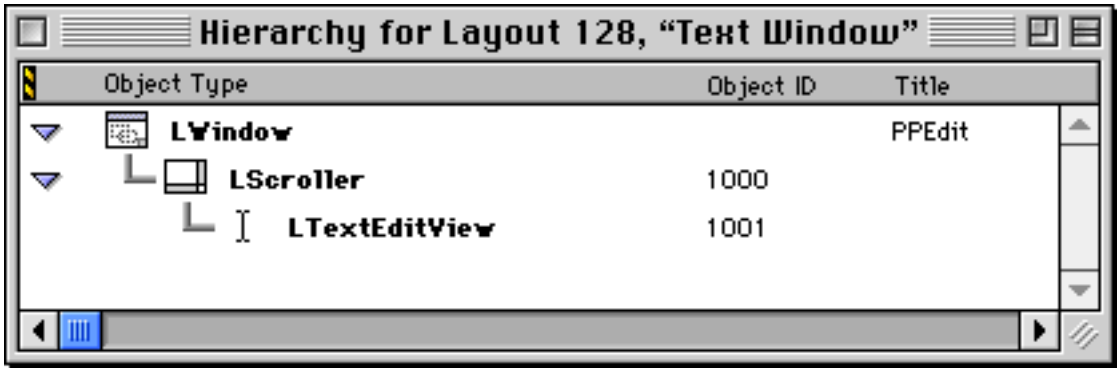
When you're through, close the property inspector window and save your work.

8. **Examine the view hierarchy.**

   In a view hierarchy, some views contain other views. This is critical in some situations, such as when you have a view inside a scrolling view. When you drop a pane inside a view in Constructor, as you did in the previous step, Constructor puts the pane inside the view hierarchically.

   To view the current hierarchy, choose **Show Object Hierarchy** from the **Layout** menu. When you do, the hierarchy window appears as shown in Figure 1.10.

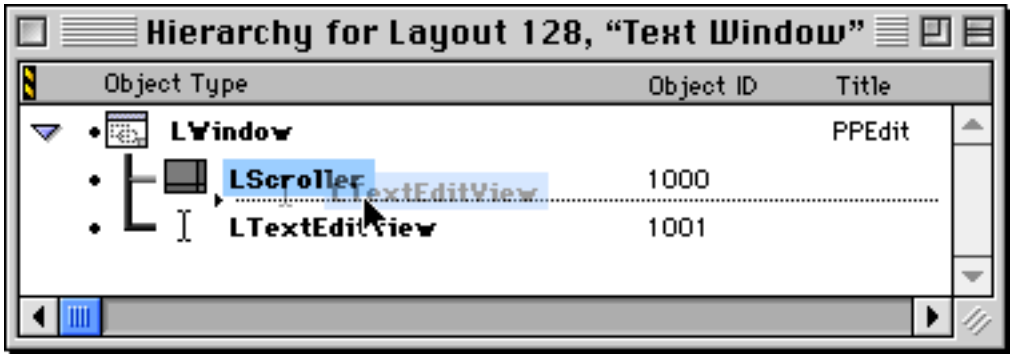
Figure 1.10 The hierarchy window



Notice that LTextEditView is "inside" LScroller and both are in the LWindow object. If your hierarchy window looks like this, you're all set. You can go to the next step.

If LTextEditView is not inside the LScroller, you can set the hierarchy manually. To accomplish this, click and drag the LTextEditView object one position to the right underneath LScroller. This can be a little tricky the first time you do it, but the secret is to put the mouse over the item inside which you wish the object to appear, as shown in Figure 1.11.

Figure 1.11 Changing the hierarchy



As you drag, a tiny black triangle and flashing line appear to indicate where in the hierarchy you are about to drop the item. When you are through, your hierarchy should look like Figure 1.10, with the LTextEditView object underneath and indented one position to the right of the LScroller object. The LTextEditView doesn't actually indent until after you finish the drag. A disclosure triangle appears to the left of the LScroller to show you that it has contents.

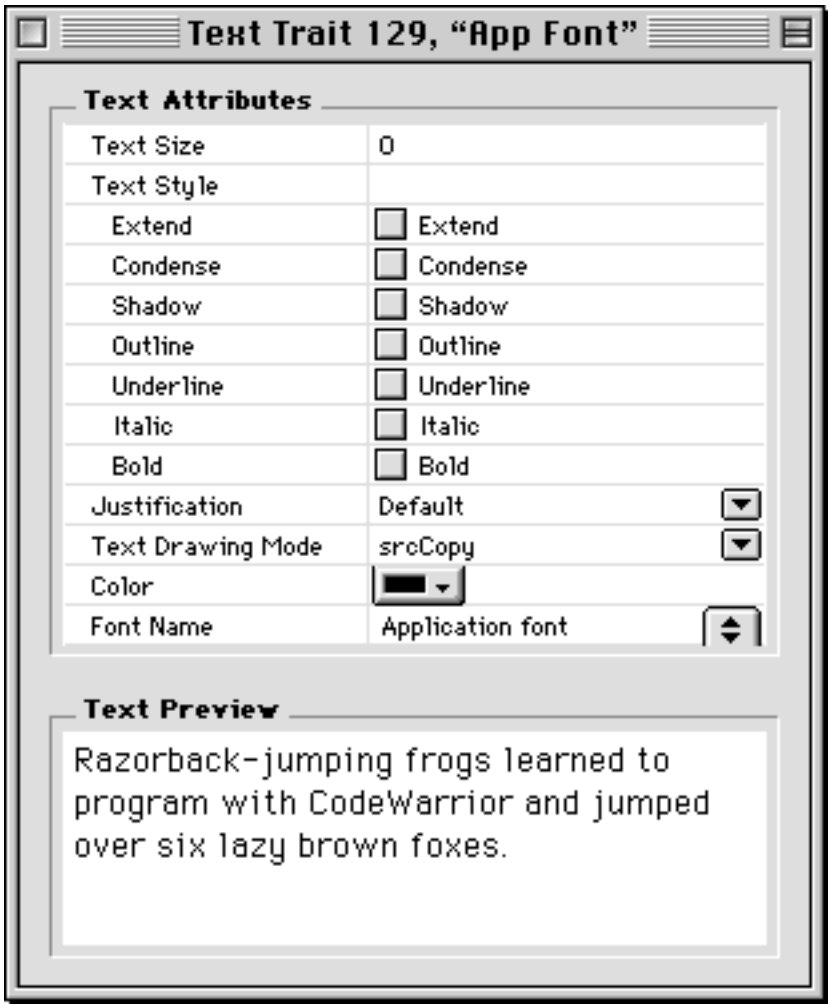
When you are through, close the hierarchy window, and save your work.

9. **Examine text traits.**

Constructor allows you to create text traits resources that you can then apply to panes that contain text.

Return to the Constructor project window, and double-click any one of the text traits resources. A window appears that allows you to edit the characteristics for the text, as shown in Figure 1.12.

Figure 1.12 Setting text traits



You don't have to make any changes to the resource, this step is just to show you what the text traits resource is. Remember that you set the text edit pane you created in step 7 so that it uses the text traits resource with the ID number 129.

When you are through studying, close the text traits window.

10. Quit Constructor.

Save your work and quit Constructor. You have just specified a complete visual interface for our simple text-editor. In the process, you created a single PPob resource and a visual hierarchy. That re-

source describes the window and all of its contents: the scrolling view and the text view.

In the next section you write the code to display that window on the screen, and use it to enter text. You may be surprised to learn that the steps you have just completed are more complicated than writing the code to create the application! Let's write the code and see.

Writing PPEdit

In code-related steps, you will encounter a feature that helps you identify what function or location in the file you are working with, and what file it is in. It looks like this:

```
function name()                                    File.cp
```

You'll see a line like this at the beginning of every step. We assume that you have opened the file and located the right spot in the code.

In the following steps you write all the code necessary to fully realize the PPEdit text-editor. Each step lists the code you must write in the step. Some code has already been provided for you. The code printed in each step usually includes some of the existing code, to make sure you put the new code in the right spot. Existing code is always in italic.

If you don't understand exactly what it is you're doing in any of these steps, don't worry about it. Just follow along. The mysteries will become clear as you work through the rest of the book.

1. **Examine the CPPEditApp.h file.**

```
class declaration                                  CPPEditApp.h
```

To make a PowerPlant program, you typically make a new application class derived from LApplication. This new class overrides certain member functions of LApplication—such as `ObeyCom-`

mand()—to perform specific program actions. In the following steps you complete the functions of CPPEditApp.cp.

If you haven't already done so, switch back to the CodeWarrior IDE and take a look at the CPPEditApp.h file, located in the Chap 01 Start Code folder.

As you see from the class declaration, the CPPEditApp class derives from LApplication. There are no new functions added to this class. However, this class overrides three functions in the base class. They are:

- ObeyCommand()
- FindCommandStatus()
- Startup()

You'll modify each of these functions in the next few steps. For now, just note their existence. When you are through, close this file.

2. **Declare a constant for the PPob resource.**

near start of file — CPPEditApp.cp

Near the beginning of the file, you'll see a comment as shown in the code below. Define a constant with the value 128 to match the ID number of the PPob resource you created earlier in this exercise.

```
// Step 2: declare a const for the PPob resource
const ResIDT textWindow = 128;
```

The correct data type is ResIDT. This is a PowerPlant-defined data type used for resource ID numbers. You'll meet it again later.

3. **Register required PowerPlant classes.**

CPPEditApp() — CPPEditApp.cp

PowerPlant must create the objects you specified in the PPob resource. There are functions in each class to create the objects of each

class. However, you must tell PowerPlant where those functions are. That's the purpose of "registering" the PowerPlant classes.

You register each PowerPlant class you need individually. (You use the same registration method when you create your own classes and objects. We'll talk about how to do that later in this book).

In the `CPPEditApp()` constructor, write the following code.

```
CPPEditApp::CPPEditApp()
{
  // Register classes required to create our window

  RegisterClass_(LWindow);
  RegisterClass_(LScroller);
  RegisterClass_(LTextEditView);
}
```

4. **Open a window at startup.**

`Startup()` `CPPEditApp.cp`

When the application launches, your application receives one of three Apple events from the Finder—open application, open document, or print document. If you launch without documents, the application calls its own `Startup()` function in response to the open application Apple event.

Add this code to the function (existing code is in italics).

```
CPPEditApp::StartUp()
{
  ObeyCommand(cmd_New, nil);
}
```

You are telling the application to perform a specific command. In this case, the command is `cmd_New`. The `cmd_New` value is a constant defined in the PowerPlant headers. It corresponds to the **New** item in the **File** menu. (We'll discuss exactly how menu commands work later in the book).

NOTE: The Startup() function is only called if the application is aware of Apple events. To ensure that it is, you should make sure that the `isHighLevelEventAware` flag (in the SIZE flags) is on in the project preferences. This flag has been set for you in this project.

5. **Respond to the New command.**

`ObeyCommand()` `CPPEditApp.cp`

Your application should obey the "New" command, and make a new window. Add this code to the `switch` statement, before the existing default case.

```
case cmd_New:
  LWindow *theWindow;
  theWindow = LWindow::CreateWindow(textWindow,this);
  break;
```

This code tells PowerPlant to make and display an LWindow object with the PPob resource ID textWindow (that you defined in step 2 to be 128). And it says that this window's commander is the application.

NOTE: PowerPlant doesn't just create the window, it creates everything inside the containment hierarchy of the window, including the scrollbars and the editing pane. PowerPlant was designed to let the PPob resource format do a lot of the organizational work, so you can design a program interactively in Constructor.

All other commands, such as Quit, are passed to the `LApplication::ObeyCommand()` function by the default case.

6. **Enable the New menu item.**

`FindCommandStatus()` `CPPEditApp.cp`

This application must have the **New** command on the **File** menu enabled. Setting a menu item's status is accomplished with the Find-

CommandStatus() function. PowerPlant calls this function for each menu item.

Add this code to the `switch` statement, before the existing default case.

```
case cmd_New:
  outEnabled = true;
  break;
```

This code means that the **New** item in the **File** menu is always enabled.

7. **Run the application.**

   That's it! Honest!

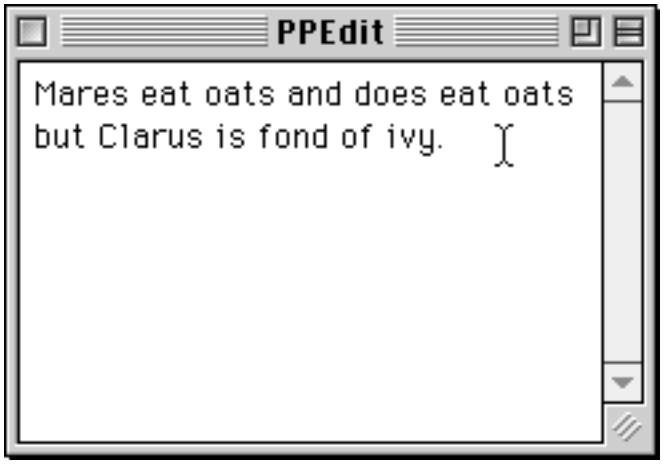
   Save your work. Then choose the Make command to build the new application. Compiling will start slowly, because the environment must search for the headers. Subsequent compiles in this project will be much faster.

   If you have any errors, carefully examine your code against the steps in this exercise, to see where you might have forgotten a detail. If you cannot find the problem, you can use the code in the Solution Code folder for this exercise.

   When the project has been successfully built, choose **Run** from the **Project** menu.

   The PPEdit application will appear, with just three menus: Apple, File and Edit. A new blank window appears as well. Click in the window and start typing.

Figure 1.13 The PPEdit Window



Look at all the functionality you get for free! You can enter text, cut, copy, paste (from outside applications as well), clear, and select all. You can display an About Box. You can create new windows, close them, zoom, resize, drag, and grow the windows. You can add more windows, limited only by memory. You can switch windows freely, and they activate and deactivate appropriately. The scroll bar works! And you did all this with about ten lines of code.

All of this behavior is a gift to you from PowerPlant. The classes that you are using—LApplication, LWindow, LScroller, and LTextEditView—all have default behavior to support these activities. That gift—the default behavior of a complete and powerful Macintosh application—is why using PowerPlant is so worthwhile.

When you are through playing with your new application, choose **Quit** from the **File** menu, and you're done. (Yet another freebie from PowerPlant.)

If you're the curious sort and you'd like to follow along with the PowerPlant code, you can do so easily. To trace the commands and keystrokes, choose **Enable Debugging** from the **Project** menu. The next time you run the application, it will invoke the CodeWarrior Debugger. You can step through each command as you choose. For more information on debugging, see the *Debugger User Guide*.

If you look at the size of your new application, you may be surprised at how big it is for how little work you did. When we build PPEdit according to the instructions, the application occupies about

175K on disk depending on which build target you use. You can make it smaller if you compile with optimizations on, and without debugging.

What Next?

Of course, there are a lot of things this little application doesn't do, like printing, or saving and opening files. We're going to be talking about those topics and much more in the chapters ahead of us.

If you are new to PowerPlant programming, and you'd like to learn more before you go off adventuring, feel free to ignore the suggestions for enhancements we're about to make. However, if you're in an adventurous mood, here are some suggestions for what you can do with PPEdit.

As you have seen, you can get a lot from just a few lines of code in PowerPlant. But there are a number of features that would make this little text editor nicer. You can try to add these features. If you do, we guarantee you'll learn a lot about PowerPlant.

Of course, because you don't know much about PowerPlant yet, things may not go right. That's often what happens when you explore unfamiliar territory. But you will be exploring and you'll remember the places you visit and the sights you see. Try these on for size.

- Make the window ready for typing when opened by using the calls `FindPaneByID()` (with the ID of the text edit pane) and `SetLatentSub()`
- Implement dynamic scrolling, where the text scrolls as you move the thumb in the scroll bar. You'll have to do a little work in Constructor (using an LActiveScroller object). You'll also need to register the LActiveScroller class.
- Implement support for Undo and Redo!
- Add Font, Size and Style menus to update the text style in the window. (LTextEditView supports multiple fonts, sizes, and styles.)
- Support opening and saving files by deriving the application class from LDocApplication.

- Add some dialogs using LDialogBox or StDialogHandler.

The first two suggestions are probably the easiest. Did you notice that when the window appears, you have to click in the window to activate text entry? Wouldn't it be nice if the window was already set for text entry when it opened? Dynamic scrolling is a breeze to add. PowerPlant provides all the functionality automatically.

We realize that at this point you may feel you don't know enough about PowerPlant to pursue these ideas. You may be saying to yourself, "What's this `SetLatentSub()` stuff?" and "How do I use StDialogHandler?"

That's perfectly normal. PowerPlant is big and powerful, and it is based upon high-level design concepts with which you may not be familiar. That situation is about to change. Before long, easy solutions that allow you to implement all of these features will be apparent to you.

How is that going to happen? You're going to read this book. In the next few chapters—the Background chapters—we're going to talk about installing PowerPlant, PowerPlant naming conventions, and the high-level design principles upon which the PowerPlant architecture is based.

After you have that solid foundation in the fundamental underpinnings of PowerPlant, we're going to take you through every aspect of application programming using PowerPlant.

Starting Your Own PowerPlant Projects

PPEdit and the other example projects in this book come already set up for you with all the files already part of the project. To start your own PowerPlant project, you would use one of the stationery projects supplied with CodeWarrior. To use the supplied stationery, choose **New Project** from the **File** menu.

The available PowerPlant stationery is shown in Figure 1.14. Choose the stationery project that satisfies your requirements and click **OK**. CodeWarrior displays the Standard File dialog for you to give your

project a name and save it. Once your project is created, you can add or remove files as required.

Figure 1.14 PowerPlant stationery

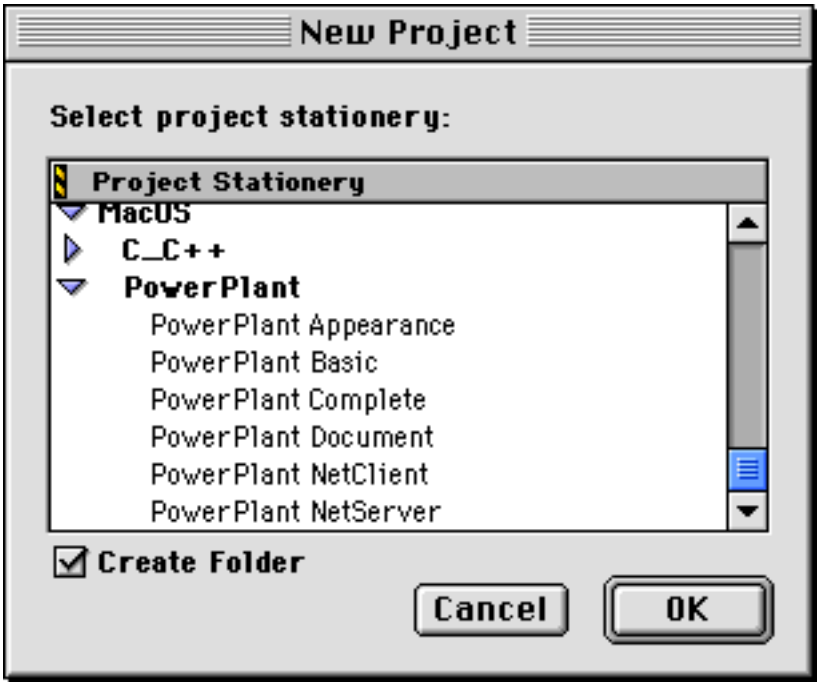


Table 1.1 describes each stationery project.

Table 1.1 PowerPlant stationery descriptions

Stationery	Description	Build Targets	Main Files
Appearance	Appearance manager savy application derived from LApplication.	68K, PPC, FAT Debug only	`CAppearanceApp.cp, CAppearanceApp.h, Appearance.ppob, & Appearance.rsrc`
Basic	Basic PowerPlant application. Application class derived from LApplication. Includes only basic PowerPlant classes	68K, PPC, FAT Debug only	`CBasicApp.cp, CBasicApp.h, Basic.ppob, & Basic.rsrc`

Stationery	Description	Build Targets	Main Files
Complete	More advanced setup of a PowerPlant Application derived from LApplication. Includes most of PowerPlant	68K, PPC, FAT Debug and Release	`CCompleteApp.cp`, `CCompleteApp.h`, `Complete.ppob`, `Complete.rsrc`
Document	PowerPlant Application derived from LDocApplication.	68K, PPC, FAT Debug only	`CDocumentApp.cp`, `CDocumentApp.h`, `Document.ppob, & Document.rsrc`
NetClient	Local network (non-internet) client application derived from LDocApplication.	68K, PPC, FAT Debug only	`CNetClientApp.cp` `CNetClientApp.h` `NetClient.ppob` `NetClient.rsrc`
NetServer	Local network (non-internet) server based application derived from LDocApplication.	68K, PPC, FAT Debug only	`CNetServerApp.cp` `CNetServerApp.h` `NetServer.ppob` `NetServer.rsrc`

Each stationery project is a fully functional application. You may want to change the main application file names to coincide with the name of your program.

For example, if you want to write a new document-based application called HyperApp, you choose the PowerPlant Document stationery and create your project. You would then change the files `CDocumentApp.cpp`, `CDocumentApp.h`, `Document.ppob`, and `Document.rsrc` to `CHyperApp.cp`, `CHyperApp.h`, `HyperApp.ppob`, and `HyperApp.rsrc`.

You are not *required* to rename any files, but it is a good idea to do.

See also The *IDE User Guide* for information on renaming files in a CodeWarrior project.

NOTE: Our hypothetical application file names are substituted for the file names used in the stationery projects throughout this book.

2

Installing PowerPlant

Before you can begin developing software with PowerPlant, you must have the necessary hardware and software. This chapter covers the details on these topics:

- PowerPlant Requirements—system requirements for development, and what a finished PowerPlant application needs to run.
- Installing PowerPlant—how to install PowerPlant, and what gets installed in the process.
- Installing Resource Templates—adding templates to ResEdit and Resorcerer so you can edit PowerPlant-specific resources.

PowerPlant Requirements

There are two kinds of requirements you should consider:

1. Hardware and system software you need to write a PowerPlant application.
2. Hardware and system software the user needs to run the final application.

Development Requirements

Memory and hardware requirements to develop applications with PowerPlant are the same as to run the CodeWarrior IDE. More RAM may be required if you run the IDE, Constructor, and your application concurrently.

See also The *IDE User Guide* for information on system requirements.

Runtime Requirements

All PowerPlant applications (programs that you create with PowerPlant) require the following minimum Macintosh setup:

- A minimum of 384 KB of available RAM. (Your particular application may need more.)
- A 68020, 68030, or 68040 (LC versions are fine) or PowerPC processor. *PowerPlant applications require Color Quickdraw and thus do not run on 68000 systems*.
- System 7.0 or later.
- For PowerPC systems, the ObjectSupportLib shared library. This library is an integral part of System 7.5 and later. ObjectSupportLib is built into Mac OS 8.0 and later.

NOTE: For earlier systems, the user must have installed the ObjectSupportLib shared library in their Extensions folder. This file is on the CodeWarrior Tools CD in several locations. You may ship it with your products without additional licenses.

Installing PowerPlant

You should follow the installation instructions provided with CodeWarrior to install the files for PowerPlant. The safest way to install PowerPlant (and ensure you aren't missing anything vital) is to use the CodeWarrior installer.

Examine the options available in the installer. Most of the Mac OS-related options install PowerPlant as part of the process. The only options that do not are the minimal Mac OS installations. Even if you choose a minimal install, you can still select the "Metrowerks PowerPlant" option in the installer.

The installer does all the rest of the work. Installing PowerPlant creates a folder named Metrowerks PowerPlant inside your CodeWarrior folder. That folder contains the following items:

- Constructor—the PowerPlant view editor

- Custom Types—folder of custom resource types used by Constructor
- Converters—folder containing conversion applications to convert older PPob files and resource types
- PowerPlant—an alias to the PowerPlant source code

PowerPlant Source Code

The original PowerPlant folder referred to by the alias is located in the "MacOS Support" folder inside the Metrowerks CodeWarrior folder.

This folder contains all the PowerPlant source code, both header and source files. They are grouped according to functionality into a series of folders such as Pane Classes, Commander Classes, and so forth. You will meet most of these classes as you read this manual.

All the PowerPlant source code is directly available to you so that you can study it, use it, see it in the debugger, and even modify it if modification is vital to your project.

WARNING! Although you can do so, you should think twice before modifying PowerPlant code. The whole purpose of an object-oriented application framework is to give you a structure that you can easily extend to suit your needs. You should rarely if ever run into a situation where modifying the actual PowerPlant code is either necessary or advisable.

The PowerPlant folder also contains the PowerPlant Resources folder. The PowerPlant Resources folder contains templates for both ResEdit and Resorcerer so that you can use those applications to edit and manipulate vital PowerPlant resources. We'll revisit this topic in "Installing Resource Templates" on page 61.

PowerPlant Documentation

The "PowerPlant Doc" folder is on the CodeWarrior Reference CD in the CodeWarrior Documentation folder. The PowerPlant docu-

mentation consists of this manual, *PowerPlant Advanced Topics*, the *PowerPlant Revealed* HTML reference, and the *Constructor for PowerPlant Guide*.

PowerPlant Advanced Topics covers a variety of significant PowerPlant issues. It is a collection of independent chapters complete with example code.

PowerPlant Revealed is a series of HTML files that covers detailed information on PowerPlant classes, methods, and member variables. *PowerPlant Revealed* can be viewed using any graphical web browser.

The *Constructor for PowerPlant Guide* provides the information necessary to use PowerPlant's view editor. You'll find it very useful as you get to know and use Constructor.

PowerPlant Example Code

The tools installer does not install PowerPlant example code. Nonetheless, there is an abundance of PowerPlant example code available to you on the CodeWarrior Reference CD. You'll use some of that code in the code-intensive chapters of this manual.

The example code is located on the CodeWarrior Reference CD in the `MacOS Examples` folder inside the `CodeWarrior Examples` folder. Inside the `PowerPlant Examples` folder there are several folders full of PowerPlant-related code.

The `PP Book Code` folder contains the code you use in this manual. We'll discuss installing and using that code in detail when we reach the first code exercise in this manual.

TIP: Also visit PowerPlant contributed class archive available at `http://www.metrowerks.com/tools/` for more classes written by other PowerPlant enthusiasts.

Installing Resource Templates

In the process of writing a PowerPlant-based application, you will create and use several unique resource types:

- PPob—specifies the view hierarchy and elements
- Mcmd—specifies menu commands
- RidL—an ID list of controls
- Txtr—specifies the text traits used by objects that display text

Constructor is the principal tool you use to create and edit PPob (PowerPlant object) resources. The PPob resource specifies the contents of a view (typically a window or part of a window), and how the contents relate to each other hierarchically. Constructor allows you to interactively create and design windows with lists, controls, pictures, and more by simply selecting items from a palette and placing them where you want them to appear in the window.

NOTE: Constructor does not generate code, it simply creates the PPob resource. PowerPlant itself contains the code necessary to build the standard objects you specify in the PPob resource.

You can also use Constructor to create and edit MBAR, MENU, and Txtr resources. Constructor creates but cannot edit RidL resources. Constructor also lets you create and edit Mcmd resources right in the menu editor.

Although Constructor lets you work directly with all the necessary resources, you may wish to use a third-party resource editor such as ResEdit or Resorcerer to manipulate these resources. We have provided Rez and Resorcerer templates for all the resources, and ResEdit templates for some of them. Follow the instructions below for the editor you want to use.

See Also "Basic PowerPlant Resources" on page 113 for more information on PowerPlant resources.

Resorcerer

The Resorcerer templates for all four PowerPlant resources are in the `PowerPlant Resorcerer TMPLs` file. Locate this file in the PowerPlant Resources folder inside the PowerPlant folder (the one that contains all the source code).

Be sure that Resorcerer is *not* running. Copy the file `PowerPlant Resorcerer TMPLs` to the "Private Templates" folder inside your Resorcerer folder. The new templates will be available the next time you launch Resorcerer. It's really that easy.

ResEdit

The ResEdit templates for three PowerPlant resources are in the `PowerPlant ResEdit TMPLs` file. Locate this file in the PowerPlant Resources folder inside the PowerPlant folder (the one that contains all the source code).

Open the `PowerPlant ResEdit TMPLs` file with ResEdit. Then open the `ResEdit Preferences` file (in the Preferences folder of your System folder). Go back to the PowerPlant file, select the TMPL resources, and copy them. Paste the resources into ResEdit Preferences, then save and close the file. The new templates will appear immediately.

NOTE: The PPob resource does not have a ResEdit template. You can't edit PPob resources in ResEdit because the layered containment hierarchy in a PPob resource is too complex for ResEdit's template mechanism.

Rez

Rez is a resource compiler that translates text files into resources. (DeRez is the corresponding decompiler). You can use Rez as a tool under ToolServer (an MPW tool available under the CodeWarrior IDE) or by using the plug-in Rez compiler that comes with CodeWarrior. The necessary information to use Rez with Power-

Plant resources is in the `PowerPlant.r` file in the PowerPlant Resource folder. You'll find this file in the PowerPlant Resources folder inside the PowerPlant folder (the one that contains all the source code).

Most programmers who use Rez typically use it for resources that have convenient text representations. This includes all resources whose data is primarily strings or numbers, but also PPob resources, which have a simple textual format. You do not need any special templates to work with PowerPlant resources in Rez.

Summary

After you install PowerPlant, locate the source files so you can find them when you need them. You might want to locate the documentation files, and copy the *PowerPlant Revealed* HTML reference to your hard drive.

Also remember to install the resource templates for the resource editor of your choice, ResEdit, Resorcerer, or both.

Now you've got some great tools lined up, and you're ready to go to work! Before you do, let's cover some PowerPlant terminology and get comfortable with how the tools are organized.

3

PowerPlant Conventions

In this chapter we discuss the style in which PowerPlant code is written, including standardized naming conventions you'll find very useful. As with all things stylistic, you are not required to follow these guidelines in your own code. However, understanding these conventions will help you understand what's happening when you are studying PowerPlant source code.

This chapter explains the following naming and coding conventions to you:

- Class and File Names
- Variable and Parameter Names
- Data Types
- Other Names
- Calling Macintosh Toolbox Routines

Class and File Names

Source files usually contain only one class. Occasionally, a file contains several classes, for example, `LStdControl.cp`.

Class names and file names begin with a capital letter with embedded uppercase letters for word breaks. The first letters of the class name or file name are a prefix, reflecting the kind of class. Table 3.1 lists the naming conventions.

Table 3.1 Source and class naming conventions

Prefix	Meaning	Example
`L`	PowerPlant library class	`LControl`
`U`	PowerPlant utility class	`UKeyFilters`
`St`	a stack-based class	`StHandleLocker`

PowerPlant uses stack-based classes to save and restore state information. Their constructors have the side effect of saving state information, and their destructors restore it. See the `StHandleLocker` class in `UMemoryMgr.cp` for an example.

Variable and Parameter Names

Variable and parameter names begin with a lowercase letter and use embedded uppercase letters for word breaks. The first letters of the variable name are a prefix, reflecting how the variable is used. Table 3.2 lists the naming conventions.

Table 3.2 Variable and parameter naming conventions

Prefix	Meaning	Example
`m`	data member of a class	`mFrameSize`
`s`	static data member	`sClickCount`
`in`	input parameter	`inCommand`
`out`	output parameter	`outMenuH`
`io`	input/output parameter	`ioParam`
`the`	a local variable	`thePane`

Pointers and handles are identified by a single-letter suffix. Table 3.3 lists the naming conventions.

Table 3.3 Pointer and handle naming conventions

Suffix	Meaning	Example
`P`	a pointer	`theStringP`
`H`	a handle	`theMenuH`

Data Types

PowerPlant declares several data types for its own use based on standard C data types.

Integer types

The type names describe the number of bits in each type. PowerPlant uses these types.

Table 3.4 PowerPlant integer types

Name	C type
`Int8`	`signed char`
`Int16`	`signed short`
`Int32`	`signed long`
`Uint8`	`unsigned char`
`Uint16`	`unsigned short`
`Uint32`	`unsigned long`
`Uchar`	`unsigned char`
`Char16`	`Uint16`
`ConstStringPtr`	`const unsigned char`

Synonyms for integer types

Type definitions that are synonyms for other integer types end with a capital T. Table 3.5 lists the most common such types.

Table 3.5 Other integer types

Name	Type
CommandT	Int32
MessageT	Int32
ResIDT	Int16
PaneIDT	Int32
ClassIDT	FourCharCode
DataIDT	FourCharCode
ObjectIDT	FourCharCode

Enumerated types

Enumerated types begin with a capital E. All constants in an enumeration begin with the same prefix that identifies the type. The prefix is separated from the rest of the constant name by an underscore. For example:

```
enum EProgramState {
  programState_StartingUp,
  programState_ProcessingEvents,
  programState_Quitting
};
```

Constants

Constants always contain an underscore, used as a word break. The underscore is never the first or last character. Embedded capitals are used for other word breaks. Usually the word or words before the underscore indicate a category or kind of constant. This is similar to the naming of enumeration constants. For example:

```
const ResIDT resID_Default = -1;
const ResIDT resID_Undefined = -2;
const ResIDT cursor_Arrow = -1;
```

Resource IDs

Constants used as resource ID numbers begin with the four-character resource type code. If the resource uses illegal characters, use an approximation for those characters. For example:

```
const ResIDT MENU_Apple = 128;
const ResIDT ALRT_About = 128;
const ResIDT STRx_Names = 128;
```

Structures

Structure names begin with a capital S. PowerPlant never uses the `struct` keyword to define a class. In PowerPlant, a struct contains only data members and never contains member functions. For example, this is the structure that LDialogBox uses to respond to clicks in dialog buttons:

```
struct SDialogResponse {
  LDialogBox *dialogBox;
  void *messageParam;
};
```

Other Names

Preprocessor macro function names end with an underscore (_), such as `Assert_()` and `ThrowIf_()`.

"Mac" embedded in a name means that the item is related to a Macintosh Toolbox data structure. "Count" embedded in a name designates a quantity, such as the number of items in a container.

The terms "Image," "Port," and "Local" used as suffixes in member function declarations relate to the coordinate system that the function uses. We'll discuss the various coordinate systems in Chapter 7, "Views."

Calling Macintosh Toolbox Routines

The PowerPlant application framework builds on the Macintosh Toolbox, it does not replace the Toolbox. Most of your programs—like PowerPlant itself—will use Macintosh Toolbox routines.

To avoid any possible name collision between a Toolbox routine and a member function of a PowerPlant class, you should always use the unary scope resolution operator (::) when you call Macintosh Toolbox routines. The PowerPlant code follows this convention almost all the time.

This practice makes it easy to distinguish Toolbox routines from member functions. Here's a code snippet from `LWindow::DoSetZoom()`.

```
SetWindowStandardState(mMacWindowP, &zoomBounds);
FocusDraw();
::EraseRect(&mMacWindowP->portRect);
::ZoomWindow(mMacWindowP, inZoomOut, false);
ResizeFrameTo(zoomWidth, zoomHeight, false);
```

The calls to `EraseRect()` and `ZoomWindow()` are Macintosh Toolbox calls. The other calls are to member functions of the LWindow class.

Notice that PowerPlant code does not use the `this->` specifier for the current object.

Summary

File, class, variable, parameter, and data type naming conventions make PowerPlant code consistent in style. This helps you understand the code more easily. Macintosh Toolbox functions are almost always called using the unary scope resolution operator. In most cases PowerPlant code does not use the `this->` specifier for the current object.

You have absorbed a lot of information in this chapter. You'll find it very useful when you start exploring PowerPlant code a little later. In the next two chapters we'll explore the power and patterns you find in application frameworks in general, and in PowerPlant in particular. You're going to master the high-level design patterns used in the PowerPlant architecture.

4

Application Frameworks

This background chapter discusses in general terms what a framework is, the kind of power it has, and the kinds of design patterns you typically encounter in a framework. This will give you an understanding of the universe within which PowerPlant lives.

This overview gives you an invaluable perspective when you work with PowerPlant code. In the next chapter you'll see how PowerPlant implements the high-level relationships that designers have found to be vital in a good, object-oriented application framework.

This chapter discusses the following topics:

- Reusable Code—a general introduction to the concept of reusable code, and how the concept has matured.
- Application Frameworks—the purpose and nature of application frameworks.
- Framework Design Patterns—the typical solutions that framework designers have created to solve common application programming problems.

Reusable Code

Programmers want and benefit from reusable code. Reinventing the wheel is not a reasonable or productive use of your time and effort. In this section we're going to talk about three different ways in which programmers have organized useful collections of reusable code. They are:

- Procedural Code Libraries
- Class Libraries
- Frameworks

Each collection of reusable code is more sophisticated and more powerful than its predecessor. Because procedural libraries are familiar to most programmers, we'll start with them.

You should realize, however, that some programmers use these terms interchangeably. You will regularly hear frameworks called class libraries, and libraries referred to as frameworks. Nonetheless, there is a distinction between each of these collections, as you'll soon see.

Procedural Code Libraries

Code libraries have been around for a long time. It is quite likely that you are familiar with one or more procedural libraries. The ANSI standard C library is an excellent example of a function-based code library.

A procedural *code library* is a collection of routines (functions, procedures, subroutines) you can use in your own code. Such a library is usually designed to serve the lowest common denominator, so the routines in the library tend to be simple and generic to a fault. The routines are usually of a "housekeeping" or utility nature. They help you take care of programming chores, such as converting lower case characters to upper case, formatting strings, performing basic math, and so forth.

There are libraries designed for specific areas of programming, and these can be very helpful in solving programming problems in that area. You might find libraries for astronomy-related code, database manipulation, and so on.

Unfortunately, in many cases the library's routines are not perfectly suited to your particular programming problem. You may find you need to modify some of the code, or replace it. Modifying library code is always a chancy proposition, because there may be hidden dependencies between the function you're modifying and the rest of the library. This is a pitfall best avoided. Moreover, replacing code defeats the purpose of the library.

Despite their limitations, procedural libraries are very useful. They have saved many programmers thousands of hours of work. The

limitations are real however, and the desire for greater flexibility and more reusable code has led to significant advancements.

Class Libraries

A *class library*, in its most general sense, is simply a procedural library rewritten in an object-based language. A class library is a collection of classes designed to be of some use to you in accomplishing your programming chores.

The various classes may not even be related to each other, depending upon the nature and purpose of the library. Like a procedural code library, the class library is usually a collection of utility classes.

The main advantage that a class library has over a procedural library is extensibility. Because of object-oriented features such as inheritance and derivation, you can easily extend the utility of the library to meet your particular programming challenge.

When the library does not solve your problem completely, you can declare your own class, inherit from the library class, and override a few functions to make it fit precisely with your particular needs. You are no longer modifying the library code directly, you are extending it.

While you are replacing part of the underlying class by overriding behavior, the mechanism is simpler and more reliable than replacing functions in a procedural library. You aren't modifying the library's internal dependencies, and you still inherit all the behavior you do not override. No generic solution is ever going to perfectly solve all programmers' problems all the time. But the class library gives you a more flexible and robust mechanism for modifying the generic solution.

So, where do frameworks fit into this picture?

Frameworks

A *framework* is a structured form of class library designed to assist in solving a specific programming problem, a concept called the *prob-*

lem domain. A framework is a collection of classes designed to form a cohesive whole aimed at the problem domain.

Because they are all aimed at the same domain, the classes in a framework tend to be more integrated and more interdependent than in a simple class library. This can make a framework both more powerful and more difficult to use.

It is more powerful because the entire collection of reusable code is aimed at a single problem. This makes everything more focused. Because the problem domain is limited, the "generic" solution can be that much more specific.

A framework may be more difficult to use because, being more interdependent, you must get your head around the entire design to see how all the pieces work together. Only then can you figure out how to use the framework to solve your own particular version of the problem domain.

No matter how well designed or focused, a framework is still a generic solution to the problem. You must take advantage of the flexibility of an object-oriented language to derive your own solution from within the bounds of the framework.

A framework may be aimed at any sort of problem domain. You might design a framework to help solve database problems, physics problems, inventory control problems, and so forth. One potential problem domain—the one we're most interested in here—is application programming.

Application Frameworks

An *application framework* is an object-based framework whose problem domain is application programming. The framework is aimed at solving the problems that application programmers encounter when writing software for a particular platform. The platform of interest to us here is the Mac OS.

In general, an application framework concerns itself with the application *interface*, not the application *content*. This is an important dis-

tinction, and one you should remember, because it drives much of the design of a good framework.

The reason is simple. The interface elements of most applications are fairly standard, and therefore lend themselves to a generic solution. The content of applications is highly variable from application to application, thus making any generic solution almost worthless.

NOTE: There are some common types of data that many applications contain, such as text or tables. You will find support for this kind of data in some application frameworks (PowerPlant among them). All the same, the principal focus of an application framework is in the visible interface.

Beyond interface solutions, the second area in which an application framework provides solutions is in the flow of control in an application. A good framework identifies and dispatches events to the appropriate elements in the application.

The application framework can usually create a completely functional—but empty—application with very little effort. The framework provides a series of default or empty behaviors that serve as hooks. You override those behaviors in your own classes when necessary to mold the framework to your specific needs.

For example, an application framework is likely to have a "Draw" behavior somewhere to draw the contents of a window (or part of a window). You override the Draw behavior to draw your unique content, but you never have to worry about when or whether the Draw behavior should be called. The flow of control and basic functionality is provided for you in the application framework.

The advantages of a powerful application framework are immediately obvious. With little or no work, you already have a complete application shell ready to be filled with your own content. In addition, because many programmers have worked on or with the framework, its code is likely to be extremely reliable.

In a well-designed framework, you may find yourself using 90% or more of the framework without change. Common problems like event parsing, menu management, and idle time processing have already been solved by expert programmers.

Their solutions are embedded within the application framework. A good application framework like PowerPlant lets you take advantage of their work and reuse their code. You can really concentrate on the particular areas where your application needs special behavior. This, after all, is a lot more fun than reinventing the wheel, and is a much better use of your time and skills.

Application frameworks are often very complex, with dozens of highly interdependent classes. Learning the ins and outs of an application framework can be a daunting task. On the other hand, the architecture of a modern, well-designed application framework like PowerPlant can make the learning process a *lot* easier. You'll see how in the next chapter, when we discuss how PowerPlant implements the design patterns you typically find in an application framework.

Framework Design Patterns

If you have never used an application framework, you are about to acquire a new and very powerful tool.

However, one of the principal hurdles you will encounter is the obscure and seemingly strange code you find hiding in the source files. It's rather like being dropped into a maze where the walls not only block your view, they reorganize themselves every time you turn around.

Although it doesn't look like it from the inside of the maze, there really is a plan to the apparent madness. The path out of the maze is not obvious when you're lost at the code level. However, it is easy to see the solution from high above the maze. When you look down on the maze, patterns can emerge.

A design pattern is an intentional, coherent system based on the interrelationship of component parts.

For example, take a look at a stone building. You are quite likely to see one or two patterns used to span a gap in a wall (such as for a doorway or a window). One pattern you'll see is the post and lintel, where two uprights support a crossbeam that carries the load. Another is the arch, where a series of shaped stones carry the load across the gap. These are design patterns in masonry.

In object-oriented programming, the components in a design pattern are the classes or groups of classes that have, over time, been found to work well together to accomplish some specific task.

All applications encounter similar problems. Although the details of solutions vary from framework to framework, most frameworks have followed established patterns that work well. These patterns are part of the architecture of the framework. They are the steel skeleton that makes it strong.

Once you understand the patterns, complex and obscure code will become clear. You will know its place in the scheme of things, and it will no longer seem strange.

You'll be visiting these patterns again in the next chapter when you see how PowerPlant solves these high-level problems.

The patterns discussed in this section include:

- Applications
- Event Handling
- Command Hierarchy
- Visual Hierarchy
- Messaging Systems
- Persistence
- Utilities

NOTE: The terminology used in this discussion matches that used in PowerPlant. You may hear these patterns called by other names in other frameworks, but the underlying pattern and functionality are essentially the same.

Applications

An application framework usually has an application object. This object is responsible for providing application-level services. It launches, initializes the environment, runs an event loop, and quits.

The application object is usually the final dispositor of events not handled elsewhere in the application. As a result, it is responsible for basic menu control, document creation, and events that are not related to any particular window or part of the application, such as displaying the About box that is a familiar part of any Mac application.

Event Handling

A Macintosh application is event driven, and a framework must handle events. You will usually find event handling and/or dispatching classes that take care of this service for you. This class or group of classes is responsible for parsing the event and dispatching control after determining the nature of the event.

Event dispatch can be a tricky business. For example, assume you receive an event that contains a keystroke, and you have several editable text fields available in a window. Which one should receive the event?

Command Hierarchy

To simplify the event dispatch problem, the application framework establishes an object hierarchy so that every object that can respond to an event is owned by some other object. Ultimately the application owns them all.

An object that can handle events and respond to commands is called a *commander*. In the chain of command, those higher up are called supercommanders and those lower down are called subcommanders. Any given commander may be both a supercommander to its own subcommanders, and a subcommander to some other supercommander.

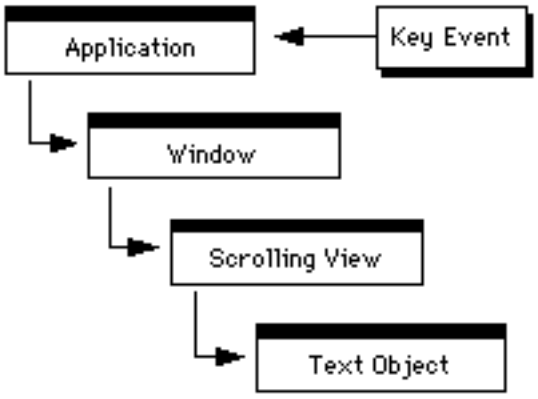
NOTE: The application object is typically the ultimate supercommander and is not a subcommander to any other object.

Typically the command hierarchy is a tree. No object may have more than one supercommander, but it may have several subcommanders. A commander may have no subcommander at all if that particular commander is a leaf on the command tree.

Even with a command hierarchy, there are still two possible solutions to the event dispatch problem. You can dispatch events from the top down, or from the bottom up.

The top-down approach starts at the application level and moves down the command hierarchy looking for an object that can handle the event (a leaf object). This approach is less common, but certainly reasonable. It has some advantages and disadvantages. Because control starts at a high level, you can process many events right there (meaning that your leaf objects can be simpler because they don't have to understand those events). On the other hand, if the event is destined for a leaf object—and most events are—dispatch must proceed down through the entire tree before the event is ultimately handled, as shown in Figure 4.1.

Figure 4.1 Top-down command hierarchy

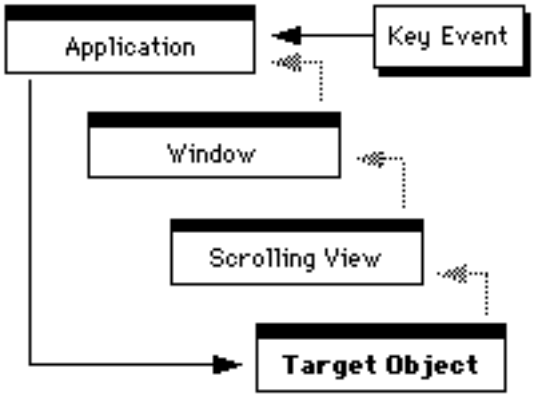


The other approach (the one you'll find in PowerPlant) is to dispatch events from the bottom up. In this design pattern, the application keeps track of a *target object*. The target object is the currently

active command object destined to be the recipient of all (or most) events. When an event is dispatched, it goes directly to the target object, which is always a commander, as shown in Figure 4.2.

If the target object is incapable of handling the event, it passes the event back up the command hierarchy to its owner/supercommander. The supercommander may handle the event, or pass it on to the next commander up the chain. Ultimately, if no object handles the event the application object receives the event back, and must handle the event itself.

Figure 4.2 Bottom-up command hierarchy



NOTE: The framework usually takes care of tracking the target object automatically. There will be times when you want to switch targets manually.

The principal advantage of the bottom-up approach is that the target object can adjust the context of the application to match the object's own capabilities. Because it gets events first the target object can set up menus properly to reflect its behavior. If the target object is a text object for example, it might enable a font menu.

This gives you tremendous flexibility. You can put a great deal of power down deep into the leaves of your command hierarchy, and let them take care of things for you. This makes the central control system a lot simpler. If you add a new kind of object, you don't have

to redesign the control system, you simply give that new object the knowledge necessary to adapt the entire application to its needs.

As an additional advantage, tracing the command chain is a lot simpler from the bottom up. Each supercommander may have several subcommanders. Figuring out which way to go from the top down is a non-trivial task. Figuring out which way to go from the bottom up is simple—there is only one path because each commander has one supercommander.

Visual Hierarchy

Because an application framework deals primarily with the visual interface, how the framework draws is an important design pattern. To make drawing as flexible as possible, an application framework establishes a visual hierarchy similar in concept to the command hierarchy.

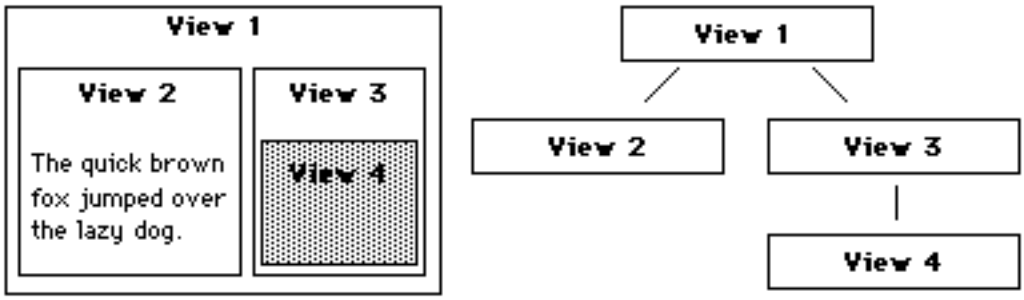
A *view* is an area within which drawing occurs. A view is *not* a window (although a window may be a view).

NOTE: You may encounter the term “pane” used synonymously with “view.” In PowerPlant, panes and views are specific classes with specific features. We’ll discuss their differences in the next chapter, and we’ll discuss their features in great detail later in this manual. For now, we’ll use the term view very generally to mean a drawing area, and nothing more.

The important feature of most application frameworks with respect to drawing is that views can form a hierarchy. A view may contain other views (subviews), which may contain other subviews, and so on.

In other words, views may have a superview-subview relationship. Like commanders, a view may be a superview to its own subviews, and a subview to its own superview. No view may have more than one superview, but it may have several or no subviews.

Figure 4.3 A visual hierarchy



The view hierarchy is responsible for maintaining the coordinate systems and the relative relationship, position, and appearance of the view objects in the chain. With a little imagination you can see that this kind of system is excellent for accomplishing tasks like scrolling, creating different views on the same data, and so forth.

In a good application framework, the view system is also responsible for maintaining knowledge of the drawing environment. On a Macintosh, this means tracking and maintaining the clipping area, graphics port, and so forth.

Setting up these features for a particular view is called setting the *focus* for that view. The view focus is an important concept. In order for view objects to draw properly, they must be the focus of attention within the operating system's graphics environment.

NOTE: Like the target object in the command hierarchy, the framework usually handles focus automatically. There will be times when you manually force a view to become the focus before drawing it.

Views are usually responsible for drawing themselves in response to messages from the control system. Drawing occurs from the top down, so that superviews are drawn before subviews. As a result, the subviews are drawn last—on top of all the superviews in the view hierarchy.

A view may also be a commander. For example, you may have a certain kind of view that displays a picture. You may want certain features enabled or disabled when the user is manipulating the picture. In order for that to happen, the view containing the picture must also be a commander so that it can respond to events.

NOTE: Do not confuse the visual hierarchy with the command hierarchy. The visual hierarchy is concerned with drawing things. The command hierarchy is concerned with handling events and commands. An object may occupy a position in either or both chains. As a result, the chains are interconnected but the chains remain distinct.

In a typical application framework you encounter a wide variety of classes derived from a base view class. These subclasses describe standard visual features of the Macintosh environment such as scroll bars, text boxes, check boxes, radio buttons, standard buttons, lists, pictures, popup menus, and more. You will encounter a bevy of these classes in PowerPlant, and you'll find them extremely useful.

Messaging Systems

You have already encountered an important messaging system, the command hierarchy. A message is received by the application, parsed, dispatched to a target object, and processed. This messaging system is vital to the functioning of an application. However, it is limited because communication only occurs vertically through the command hierarchy, and it only involves events.

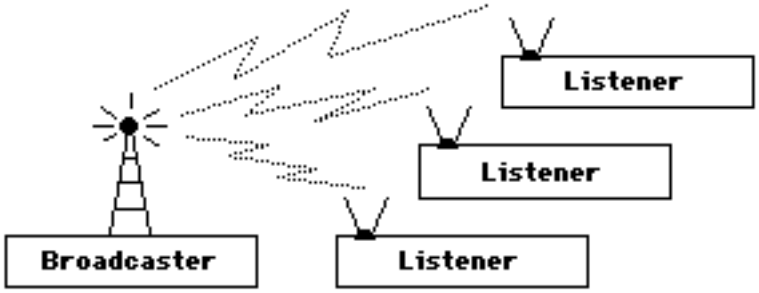
In many cases you will need objects that communicate with each other when they are not part of the same chain of command, or the same view hierarchy, or when the message involved is not an event.

Framework designers have created a wonderfully flexible messaging system to solve this problem—the broadcaster-listener system (also known as the notifier-responder).

A *broadcaster* is an object that sends a message on certain occasions. For example, a check box may send a message that it has been clicked. The nature of the message may vary depending upon circumstances. In a typical message, a control will broadcast its current setting to all its listeners so that they may respond accordingly.

A *listener* is an object that listens for messages. Depending upon the messages it receives, it may or may not respond with some action. For example, if a listener receives a message that a particular check box has been clicked and is now on, the listener may activate some new feature in a window or within the application.

Figure 4.4 Broadcaster and listener relationships



The flexibility in this system comes from the fact that the broadcaster has a dynamic list of listeners to whom it sends messages when appropriate. You may add or remove listeners at any time. The broadcaster doesn't need to know the nature of the listener, or what the listener will do in response to the message. Likewise the listener doesn't need to know anything about the nature of the broadcaster. The listener must understand the nature of the *message* in order to respond appropriately, but not the nature of the broadcaster.

This loose coupling between broadcaster and listener keeps the entire messaging system extremely flexible. You'll use it extensively when you work with PowerPlant.

Persistence

Persistence refers to the concept that some data (or more precisely, objects that contain data) should continue to exist even when the application is not running. To do that—to be persistent—this data must be stored on a medium other than volatile RAM.

A common solution is to use file and document classes to support reading and writing data from disk, and stream classes to help move that data around.

A *file* is stored data. An application framework provides classes that help you manipulate files in the standard fashion, with behaviors to open, close, revert, save, and print.

On the Macintosh, a file's contents are typically displayed in a window. In the PowerPlant architecture, the *document* class is the link between a file and a window. It connects a specific file to a specific window, and keeps track of whether the contents of the data have been changed. The document also provides support for printing the data.

A *stream* is an ordered series of bytes of data. Streams are extremely useful in a variety of programming tasks, particularly the transfer of data from one place to another (such as saving data to a disk or sending data over a network).

Files and streams, linked to windows by a document, work hand in hand to make the process of saving data to disk and reading the data from disk as painless as possible. The document keeps track of which window is attached to which file, and the physical location of the file. You use the stream classes to write data to or read data from the associated file.

Taken together, the document/file/stream combination can insulate you from the details of managing file I/O at the operating system level.

Utilities

Any good application framework provides a variety of services to you to make your programming chores easier. In addition to the command, view, and messaging systems, the framework also provides utility-level services.

The precise services will vary from framework to framework, but you are likely to encounter classes designed to help you with:

- linked lists
- sorted lists
- dynamic arrays
- memory management
- menu management
- periodical events
- idle time processing
- string manipulation
- window management
- exception handling

Learning about all the features available to you in a major application framework is a non-trivial task. There are close to 2,000 functions in PowerPlant. The good news, as you may recall from earlier in this chapter, is that you get to inherit the vast majority of them without ever seeing or being concerned with the functions in any way.

You'll discover that there are a few important places in PowerPlant where you must derive classes, a few common functions you must override, and a few places where you must provide new functionality. These places are well known and easily identifiable. Once you know what they are, you can create a simple and unique application fairly easily.

Summary

In an ever-improving attempt to create reusable code, code designers created procedural libraries, class libraries, and ultimately frameworks aimed at specific problem domains.

Application frameworks provide a generic solution to the significant coding problems faced when writing an application.

The design patterns used in application frameworks include features that support event handling and dispatch, command flow control, the visual display of standard interface elements, inter-object messaging, file I/O, and utility functions.

The PowerPlant universe is expanding before your eyes. You know what an application framework is, and what it does. In the next chapter we'll talk about the specific PowerPlant classes that implement the design patterns we have talked about here.

After that we'll start writing real PowerPlant code. In the process we'll introduce you to all the places where you will usually interact with PowerPlant. Soon you'll be able to use PowerPlant's immense power easily and effectively.

5

PowerPlant Architecture

In this background chapter we are going to talk about the structure of PowerPlant on three levels: object-oriented design, classes, and resources.

The first section of this chapter introduces you to some of the critical design decisions that make PowerPlant much easier to manage than your typical Macintosh application framework.

Because PowerPlant is a carefully designed application framework, you will see a clear relationship between the design patterns discussed in the previous chapter, and the important classes we're going to introduce to you here. We're also going to visit the basic PowerPlant resources so that you know what they are, what they are for, and how to use them.

Finally, we're going to talk about the steps you would follow as you develop a PowerPlant application.

The principal topics in this chapter are:

- Design Principles—the underlying "philosophy" behind PowerPlant.
- Framework Implementation—the classes that create the main features of the PowerPlant framework.
- Basic PowerPlant Resources—the resources you create for a PowerPlant application.
- PowerPlant Development—what it's really like to develop a PowerPlant application.

Design Principles

PowerPlant is a third-generation Macintosh application framework. Its designers learned a lot from the problems encountered in other application frameworks. PowerPlant has always been a pure C++ application framework. It hasn't inherited any baggage from earlier versions originally implemented in languages lacking object-oriented features. PowerPlant uses fundamental object-oriented behavior such as derivation and inheritance to build its class hierarchy and add functionality and behavior to the framework.

The design of the PowerPlant application framework is guided by the following fundamental principles:

- Multiple Inheritance—a mix-in architecture based on multiple base classes.
- Factored Design—classes are as independent as possible.
- Factored Classes—classes are as small as possible.
- Factored Behavior—individual behaviors within a class are carefully separated into simple, component parts.

These principles taken together ensure that the PowerPlant application framework is small enough that you can learn it quickly, yet powerful enough that you can create full-featured, world-class applications. Let's look at each of these principles to see how they affect the PowerPlant framework, and why they make PowerPlant easier to learn and use.

Multiple Inheritance

PowerPlant takes full advantage of C++'s support for multiple inheritance. As a result, the class hierarchy in PowerPlant is a series of small, interconnected trees rather than a monolithic monster tree.

Mix-in and base classes

PowerPlant has a large set of classes whose sole purpose is to be mixed into other classes by multiple inheritance. Among the common PowerPlant base or mix-in classes are:

- LCommander—for command objects.
- LPane—for visible objects.
- LBroadcaster—for all objects that broadcast messages.
- LListener—for all objects that listen for messages.
- LPeriodical—for objects that should receive time on a regular basis.
- LAttachable—for objects to which attachments can be connected.
- LModelObject—for AppleEvent support, scriptability, and recordability.

Several important PowerPlant classes inherit from one or more of these mix-in classes to add functionality where and when necessary. For example the LControl class (which encapsulates control object behavior) not only inherits from LPane (because it is a visible object), but from LBroadcaster so that any control can also broadcast a message when something happens to it.

LPeriodical is another good example. Any object that should receive attention on a regular basis inherits from LPeriodical. For example, the LTextEditView class inherits from LPeriodical because it must keep the cursor blinking when it is active.

In subsequent chapters you'll learn all the details about how these objects work. The beauty of using multiple-inheritance in a mix-in architecture is that understanding the framework becomes much easier. In a very real sense, PowerPlant can mix and match classes to combine them into powerful subclasses with a rich feature set.

A mix-in example

For example, consider the class, LEditField. This class encapsulates the behavior of an editable text field like you commonly see in a dialog box. LEditField is part of the LPane hierarchy. In addition, it inherits directly or indirectly from:

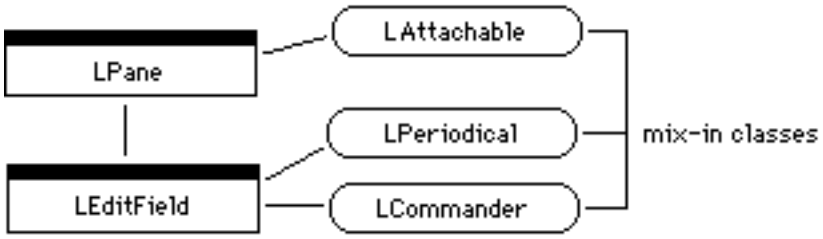
- LCommander—to respond to typing and to the commands in the **Edit** menu.
- LPeriodical—to flash the cursor when the field is active.

- LAttachable—because you may want special things to happen when an editable text field is present.

TIP: Attachments are an extremely powerful feature of PowerPlant that we'll discuss later on in this manual.

Figure 5.1 illustrates how LEditField inherits from LPane, from two mix-in classes directly, and via LPane from LAttachable.

Figure 5.1 LEditField inheritance chain



NOTE: In class hierarchy diagrams in this manual, abstract classes have a grey bar across the top instead of a black bar. There is no example of an abstract class in this diagram. Mix-in classes are represented by rounded rectangles (whether abstract or not). Identifying any particular class as a mix-in class is arbitrary, and based upon the most common way in which the class is used for derivation. If the class is used as a base class by classes in different class hierarchies, then we have identified it as a mix-in class.

In addition to making the entire framework easier to understand (because behavior is factored out into mix-in classes), the fact that PowerPlant uses multiple inheritance has an even bigger advantage for you as a programmer—enhanced code reusability.

When you write PowerPlant applications, you *will* create your own classes derived from the PowerPlant framework. Because common types of object behavior have been intelligently grouped into a se-

ries of relatively small and distinct base classes, you can simply add the desired behavior to your own class without inheriting a tremendous quantity of unnecessary baggage.

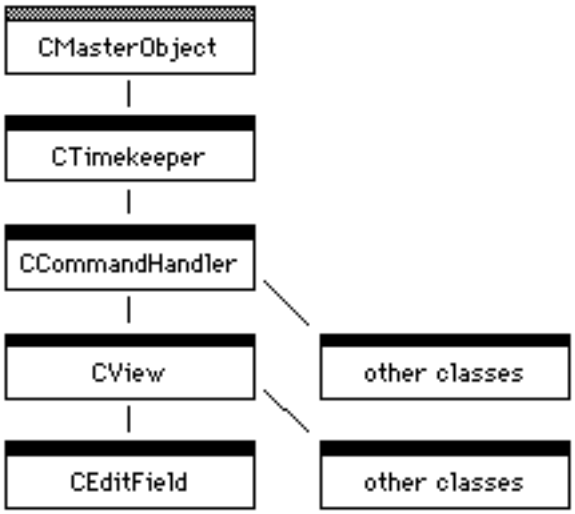
To understand how this works, let's take a look at what happens if you derive a class based on a single-inheritance hierarchy.

A single-inheritance example

In a single-inheritance hierarchy, commonly-used behaviors must appear high in the chain so that those classes that need the behavior can inherit them. Features tend to be piled onto classes not because *all* classes need them, but because *some* classes need them somewhere down the chain.

Consider how the editable text field would be implemented with single inheritance. The text object must draw itself in a window, so the text field class derives from a class in the view system. The text object should respond to mouse-clicks and typing, so the text field class also derives from the command system. Of course, that means that the view system must be part of the command system! Editable text needs a blinking insertion point, so the text object also needs to handle periodic tasks. That feature must be added somewhere up the chain, probably in the command system.

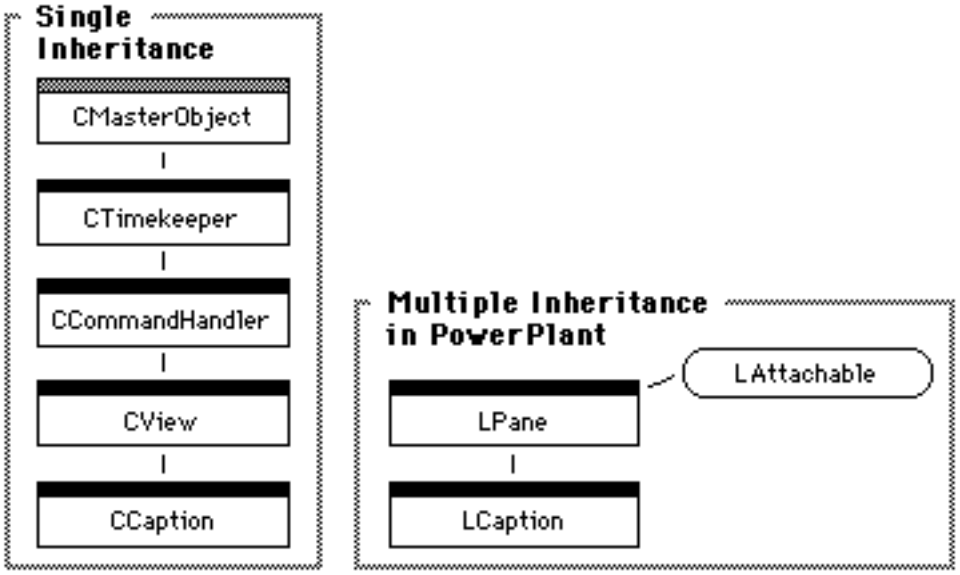
Figure 5.2 Single inheritance for a hypothetical edit field class



So, what's the problem? You've got all the same behaviors, just what's necessary and no more. It works fine for one kind of object, but there are many kinds of objects in an application.

Imagine you want a caption object that simply displays one line of text. The caption object doesn't respond to commands or perform periodic tasks, it only needs to draw itself. In the single inheritance design, the caption class would derive from the display classes, which are derived from the command classes and the periodical classes. The innocent caption class ends up inheriting a lot of baggage it will never use. When you look at that class in a browser, all sorts of spurious and useless functionality appears. You have to hunt through all the useless parts to find the behavior you need.

Figure 5.3 Single vs. multiple inheritance for a caption class



Using multiple inheritance as a principal design element in PowerPlant does not automatically make every class simple. But it does mean that any given class is far more likely to have just the behavior it needs. Some classes are still very complex, with dozens and even hundreds of behaviors. Nevertheless, without multiple inheritance the situation would be that way for almost every class, even those that are inherently simple.

Factored Design

PowerPlant works on the principle that isolating classes from one another reduces complexity and enhances code reusability. Multiple inheritance and the mix-in architecture are the principal reasons why PowerPlant can implement its second design goal—to keep classes as independent as possible.

Beyond the important base classes cited in the previous section, PowerPlant has a large collection of small base classes that you can use in a whole variety of circumstances—without using any other part of PowerPlant!

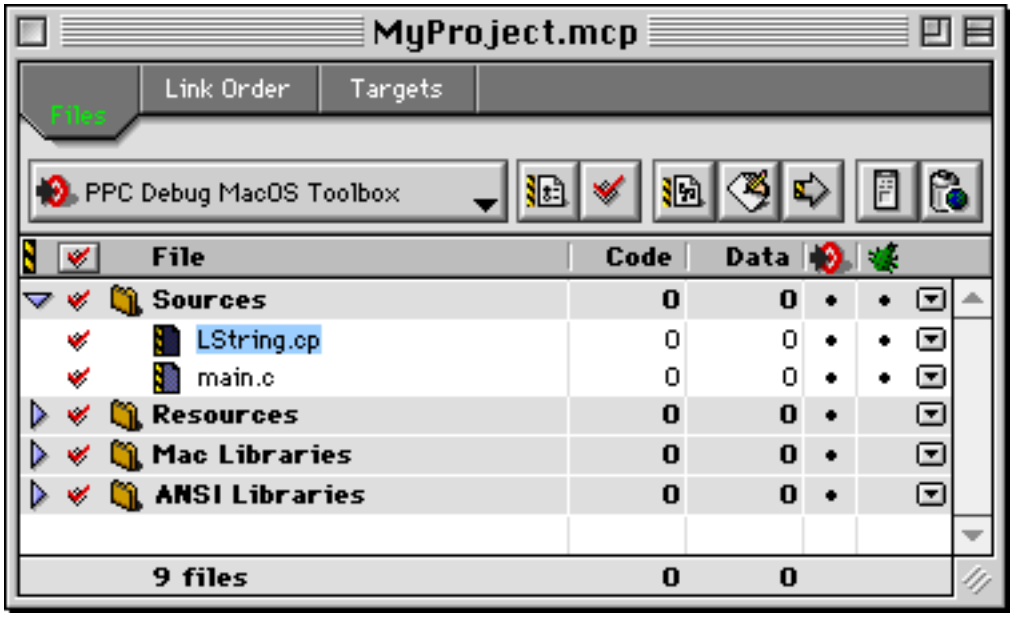
PowerPlant classes refer to as few other classes as possible. For example, the LMenu and LMenuBar classes refer to each other, but to no other classes. If you want to take advantage of PowerPlant's menu creation services, you can use those classes in your own projects without using any other part of PowerPlant. Not bad!

This design excellence makes PowerPlant a treasure trove of reusable code. Here are some more examples.

The LBroadcaster and LListener classes refer only to each other and the LArray and LArrayIterator classes. If you would like to implement a messaging system in your project, you can use these classes, and you're on your way. There's no need to inherit views, panes, commanders, or any of the other trappings of a powerful application framework.

PowerPlant includes a powerful class for string manipulation, LString. It is a complete, stand-alone class. You don't have to use any other part of PowerPlant to have access to the power of LString, as shown in Figure 5.4.

Figure 5.4 Using LString



If you need to maintain dynamic lists in your project, you can use the LArray and LArrayIterator classes independently. If you want to filter keystrokes before processing them, check out the UKeyFilters class which has a variety of filters ready for your use. Interested in some powerful debugging code? Examine the UDebugging class and UExceptions.h. The list goes on and on. There are, quite literally, dozens of useful classes that have been purposely designed to be completely or almost completely independent of the rest of PowerPlant.

Of course, in an application framework some dependencies are unavoidable. You'll find these most often in the visible objects in the view hierarchy. For example, the classes that describe control objects assume that they exist in a containment hierarchy headed by a window or equivalent object.

All the same, you can see the difference between a monolithic application framework that requires you to buy the entire store every time you want to use one tool, and a well-designed application framework based on multiple inheritance and a factored design

where you can take the tools you need and leave the rest of the hardware store behind.

Factored Classes

The third principle that guided the PowerPlant designers is that behavior should be placed as low as possible in the class hierarchy.

This process goes hand-in-hand with the overall factored design of PowerPlant we discussed in the previous section. The distinction between a factored design and factored classes is simple. A factored design looks at the problem domain horizontally. What parts of the problem can we separate from what other parts? The process of factoring class behaviors is vertical. How far down in the chain do we place a particular behavior that belongs in this hierarchy?

A careful analysis of the design of a typical Macintosh application identifies the behaviors that various aspects of the application must provide. What does a window do? What does a scroll bar do? What does a radio button do?

By analyzing the behavioral demands on the various identified objects in the system, the designers of PowerPlant have carefully factored behavior so that it appears only when necessary. General behavior appears early on in base classes, so it can be inherited by all those subclasses that need it. Behavior that is more specific to a certain kind of object doesn't appear in a class declaration until necessary.

This doesn't mean that some classes aren't large and complex. The class that defines window behavior for example, LWindow, is quite complex. It has more than 60 member functions declared in the class, plus a couple of hundred more inherited from a variety of base classes.

NOTE: Don't let LWindow scare you. Most of the functions are for internal use, and you won't use them directly. In PowerPlant, many if not most classes are small, with only a handful of member functions of their own.

However, because the designers paid attention to the underlying design, this complexity is easier to manage. Even in a complex class like LWindow you'll be able to recognize that certain member functions come from the LCommander base class, others from the LView base class, and still others from other base classes from which LWindow inherits. This logical structure allows you to break down even the most complex class into its constituent parts, making the whole that much more understandable.

Class Implementation Details

In the process of actually writing the code for these well-considered classes, the PowerPlant authors also made some code-level decisions. To ensure that all derived classes have access to the protected members of their base classes, base classes are always public, and all derivation is also public.

There are very few private data members or member functions. Almost everything in PowerPlant has either public or protected access.

Generally, the private member functions are those that are called from within constructors. In C++, if you call a function from within a constructor, it is not treated as a virtual function call. Overriding such a function in a subclass has no effect—it will not be called. Therefore, we make such functions private, so that you will not mistakenly think that you can override such functions.

Of course, like any class hierarchy, PowerPlant uses function overriding extensively. Function overloading is not so common. PowerPlant uses function overloading for constructors, and in a few other cases where the designers deemed it advisable. The only exception to this rule is the LString class and its derived classes. These classes use both function and operator overloading extensively.

For beginners: Function overriding occurs when a derived class declares a function with the identical signature as a virtual function in the base class (i.e. same return type, function name, and parameter list). Function overloading occurs when (usually within a single class) two functions have the same name, but a different signature (different parameters). Operator overloading occurs

when a class declares a replacement behavior for a standard operator, such as the addition operator (the + sign). For example, in the LString class, the + operator is defined to mean "append one string to another," as opposed to adding two numbers together.

Factored Behavior

The final principle that guided the creators of PowerPlant is that complex behaviors should be factored into simple, constituent parts. Once again, like the emphasis on factored classes and a factored design, factoring behaviors into their component parts continues the trend toward small, simple building blocks that you see throughout PowerPlant.

To implement this principle, member functions that affect an object are usually split into two parts. We'll call them the setup part and the action part.

The setup part handles any state-testing or adjusting that must happen before the action takes place, and restoring any state after the action takes place.

The action part implements the actual desired behavior.

The name of the member function that handles the setup part is usually the name of the action; for example, `Draw()`. The name of the member function that handles the action part is the same as the general function with the word "Self" added; `DrawSelf()` for example.

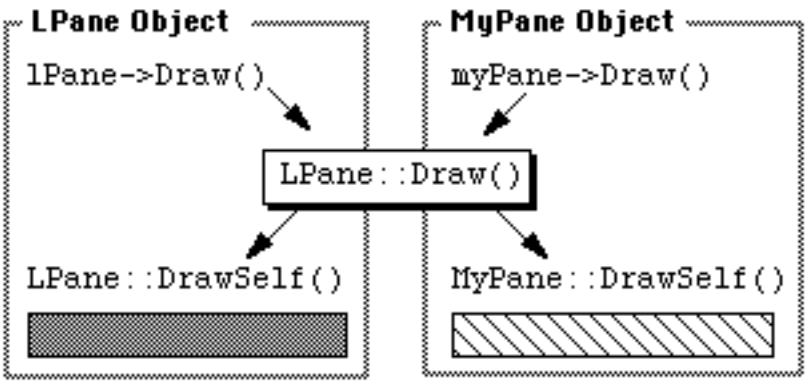
When you work with PowerPlant, you will find that you rarely if ever call a "Self" routine, but you regularly call setup routines. Conversely, you rarely if ever override a setup routine, but you regularly override the action or "Self" routine.

An example will help here. The LPane class declares two member functions, `Draw()` and `DrawSelf()`. Together these functions draw the contents of the pane. If you derive a class from LPane, you would typically call `Draw()` and override `DrawSelf()`.

The `Draw()` function makes sure that the pane is visible and that its coordinate system is set up. Then it calls `DrawSelf()`—the action routine. You have to do the setup work no matter what you draw. Setup work is usually constant so it rarely needs to be replaced, but it almost always has to be called. You can't skip the setup, but you don't normally need to replace it.

The action, on the other hand, will vary. The `DrawSelf()` function does the actual drawing, and because each pane varies you will certainly override it in a class derived from LPane. (In fact, the `LPane::DrawSelf()` function does nothing!) However, you shouldn't call `DrawSelf()` directly because the necessary setup work won't be performed and "unexpected results" will occur.

Figure 5.5 Calling Draw()



Because the PowerPlant designers factored these two behaviors into separate functions—setup and action—you don't have to write housekeeping code every time you override a function for a derived class.

You will find this type of factoring throughout PowerPlant. As you work with the code you will become familiar with what functions perform what behaviors, what functions you commonly override, and what functions you commonly call. You'll start working with code intensively in the next chapter.

Before then, however, let's take a quick tour of the actual PowerPlant classes you'll be working with. In the process, let's keep an eye

on how they fit into the overall application framework design we talked about in the previous chapter. By the time you finish this chapter, you will have a really solid foundation on which to build your PowerPlant expertise.

Framework Implementation

When we start working with the different PowerPlant classes in detail, we must present them to you in piecemeal fashion. You can't do everything all at once. In addition, the remaining chapters in this book are task-based. That is, we're going to be talking about how to accomplish particular programming tasks, and discuss what PowerPlant classes you use in that context.

This section gives you a class-based and relatively brief introduction to the various classes you'll meet in PowerPlant. This will give you an idea of what the classes are, and how they fit into the overall application framework. As a result, when you encounter them later on in this manual (from a task-based perspective) you'll know how the work you're doing fits into the big picture.

This section is organized exactly like the discussion of application frameworks in the previous chapter. Instead of theoretical design patterns, we're going to be talking about real classes. The discussion is divided into:

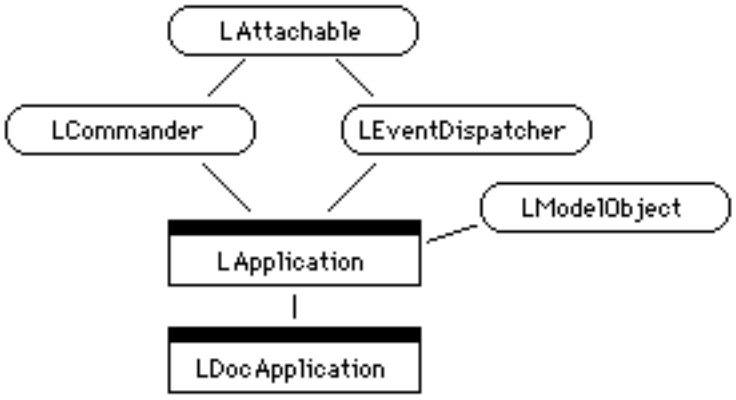
- Application Classes
- Event Classes
- Commander Classes
- Visual Classes
- Messaging Classes
- Persistence Classes
- Utility Classes

This is not all of PowerPlant. Beyond these groups of classes, there are several more classes for advanced programming issues like threads, drag and drop, scriptability, tabular data, and so forth. These additional classes are not covered in this book.

Application Classes

Applications are the core of PowerPlant programs. There are two application classes in PowerPlant, as shown in Figure 5.6.

Figure 5.6 PowerPlant application classes



LApplication encapsulates the common behavior of an application. You create a single object of this class. The application object manages the execution of an application program. It handles start up and shut down, runs the main event loop, executes application-level events (using LEventDispatcher), handles Apple events, updates menus and the menu bar, and adjusts the cursor.

LDocApplication inherits from LApplication. Additional member functions support opening, closing, and printing documents. Because most Macintosh applications store data in files on disk, you'll most likely use LDocApplication as the basis for your own application.

Typically you will derive your own application class based on one of these two classes, and override several member functions to create your application's unique behavior, handle your own menus, open, close, and print your own documents, and so forth. The functionality provided by PowerPlant for free includes initializing the environment, running the event loop, and quitting the application.

The main event loop retrieves events, and passes them on to LEventDispatcher for parsing and dispatch. The main event loop also

distributes time to objects that need attention every time through the event loop.

The application is also the top of the command chain described below in the Commander Classes section. Any event not handled by some subcommander will reach the application, where you can either process or ignore it, as appropriate. In other words, the application gets the last chance at the event.

Event Classes

LEventDispatcher takes care of all event processing after the application retrieves the event in the main event loop. Event dispatch is a fairly constant process, and in many cases you won't have to override any functions in LEventDispatcher. For the most part, PowerPlant gives you event dispatch for free.

NOTE: Unless you use the Mac Toolbox directly for tasks such as running a dialog box with `ModalDialog()`, *all* event processing goes through the main event loop and LEventDispatcher, even for modal dialogs.

LEventDispatcher is also responsible for adjusting the cursor, distributing time to idle-time processes, and initiating menu updates before displaying menus.

Commander Classes

LCommander is the base class from which all commander objects inherit. It has functions for command chain maintenance (changing supercommanders, or adding or removing subcommanders). Each commander can also manage the target object with member functions to set the target, be the target, not be the target, and so forth.

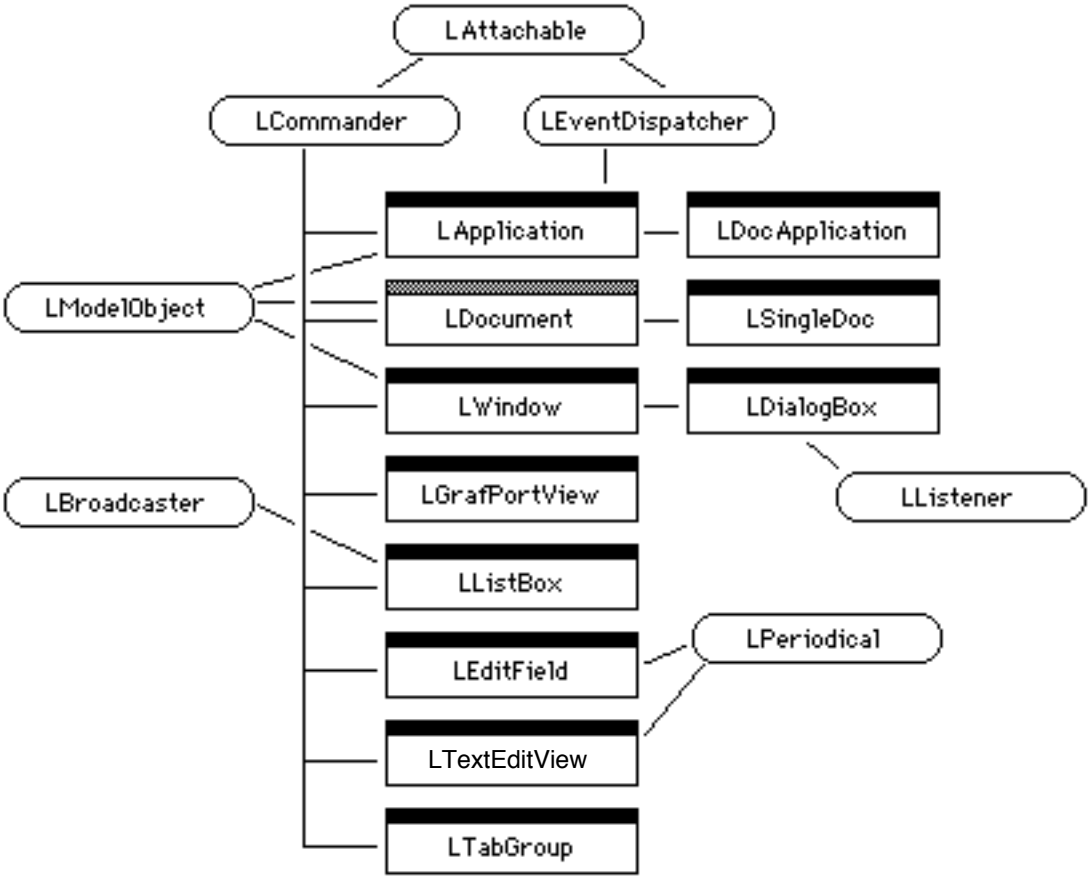
For beginners: If you are not familiar with the concept of a target, see "Command Hierarchy" on page 80.

A commander may be on or off duty, and has functions to manage the duty state. This is an important concept, because an off-duty commander will not receive or respond to events. When off duty, however, it is important to keep track of which subcommander (if any) was the target object the last time this particular commander was on duty. Then, when this commander resumes duty, the framework can activate the correct subcommander as the target object. This is called the **latent subcommander**. Each commander has the ability to set or change its latent subcommander.

Commanders are responsible for managing the state of menu items while they are the active target object or in the active chain of command. Each commander has a `FindCommandStatus()` function. When the application wants to know the state of a menu item, it calls the target object's `FindCommandStatus()` function. In response, the target object tells the application whether that particular item should be enabled, disabled, have a check mark, and so forth. If the target object does not concern itself with a particular menu item, it passes the message back up the chain of command. You will become very familiar with and override this function regularly.

Finally—and perhaps most importantly—each commander has member functions to respond to commands. You will become very familiar with the `ObeyCommand()` member function, and override it regularly as well.

Figure 5.7 LCommander hierarchy



NOTE: LDocument is an abstract class. Some of the classes that inherit from LCommander also inherit from other classes, such as LPane, LView, or LControl. In this diagram we emphasize LCommander. In other diagrams we'll highlight the LPane, LView, and LControl hierarchies and treat LCommander as a mix-in class.

There are several classes in PowerPlant that inherit from LCommander. Among them are:

- LApplication and LDocApplication
- LDocument and LSingleDoc
- LWindow and LDialogBox

- LGrafPortView
- LEditField
- LListBox
- LEditField
- LTextEditView
- LTabGroup

In addition to these PowerPlant classes, you are likely to derive your own classes from LCommander (or one of its descendants), when your object needs the ability to respond to commands.

Visual Classes

Classes designed to create the visual interface form the most complex and numerous group of classes in PowerPlant. That's not surprising, because the primary purpose of an application framework is to create the visual interface.

For beginners: If you are not familiar with the concept of a view, see "Visual Hierarchy" on page 83.

We're going to be very careful with terminology here. A "view" is a generic concept in application framework design, as described in the previous chapter.

In this section we're going to talk about two important PowerPlant classes, LPane and LView. Objects of the LView class are frequently referred to as "views," and in fact that's what they are. But a view *object* is a very special and distinct item, in addition to being a "view" in the generic sense.

In the rest of this chapter we will refer to views (meaning the framework concept) and LView objects (meaning any object derived directly or indirectly from LView). In subsequent chapters we leave the general discussion of frameworks behind, and we will use the term "view" more loosely to mean both the concept and the object. Context will tell you which meaning is intended.

The LPane class

In PowerPlant, the fundamental visual class is *LPane*. Every area you draw in, and everything you draw, is an LPane derivative. There are about 30 classes derived from LPane to handle different kinds of visual elements. There are pane-based classes for text, pictures, lists, tables, controls, and much more. We'll discuss all the details about panes and the various kinds of pane classes in the next chapter.

The LView class

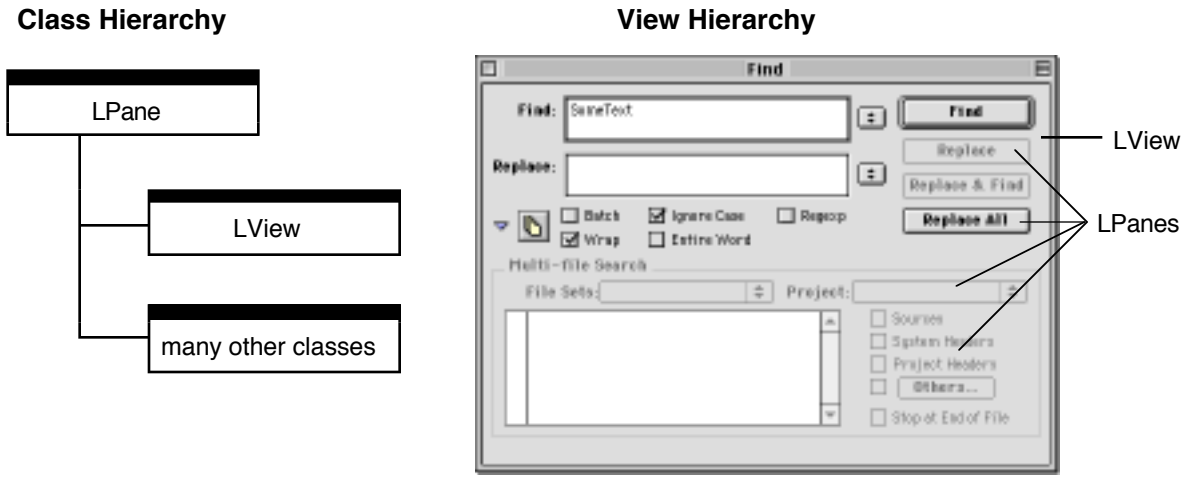
Recall from our discussion of views in application frameworks that views are typically arranged in a hierarchy. PowerPlant is no exception. The LView class forms the basis of the visual hierarchy.

An *LView* object is a pane that can contain other panes. LView inherits from LPane in the class hierarchy. However, in the visual hierarchy LView objects come first and contain all the panes that you draw. A simple way of looking at this relationship is that an LView object is a container. The panes are the contents. (Remember that an LView object can contain another LView object, creating a hierarchy of arbitrary depth).

In general, then, objects that derive directly from LPane (with the exception of LView) are those that you actually draw—objects like radio buttons, check boxes, captions, icons, and so forth. Objects that derive from LView are usually places where you draw things—objects like windows, dialog boxes, grafPorts, text views, scrolling areas, and so forth.

Figure 5.8 illustrates the different hierarchies. The simplified class hierarchy shows that LView inherits from LPane. By contrast, the topmost object in the view hierarchy is a type of LView. The contents of the window is a series of panes of various types including edit fields, captions, popup menus, check boxes, standard buttons, and so on. The scrolling list in the bottom part of the window is a scrolling view inside of the window view.

Figure 5.8 LPane and LView class and view hierarchies



Because it inherits from LPane, LView has all the behaviors of LPane. In addition, LView has behaviors for managing the view hierarchy by adding and removing subpanes. In PowerPlant terminology, when an LView object contains other LView objects or panes, it is a superview. Its contents (regardless of whether they are in fact panes or LView objects) are called subpanes.

NOTE: You create the view hierarchy in Constructor. In a typical application the view hierarchy doesn't change. Some applications modify the view hierarchy at runtime.

Messaging Classes

Inheriting from *LBroadcaster* gives an object the ability to broadcast a message. A broadcaster has a list of listeners. This simple class has functions to add and remove items from the list of listeners, and to broadcast a message.

In PowerPlant, classes that descend from LControl are all broadcasters, as is LListBox. When you create your own classes you can inherit from LBroadcaster when necessary.

Inheriting from *LListener* gives an object the ability to receive a message from a broadcaster. This simple class has a `ListenToMessage()` function that receives and can respond to messages received from broadcasters.

You will work with broadcasters and listeners often while writing PowerPlant code. We'll save the details for Chapter 8, "Controls and Messaging."

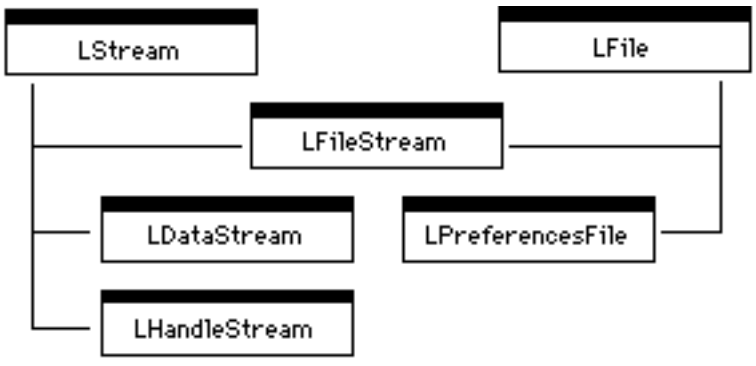
Persistence Classes

PowerPlant provides built-in support for files and streams. The *LStream* class is the basis for stream operations. It has subclasses to handle pointer-based and handle-based streams. The LStream class provides functions for reading and writing data, as you would expect.

LFile is the principal file-related class. It provides functions for opening, closing, reading, and writing both the data and resource forks of a file.

The *LFileStream* subclass inherits from both LStream and LFile, allowing you to stream data into or out of a file. We'll work extensively with these classes in Chapter 13, "File I/O."

Figure 5.9 Stream and file class hierarchy



Utility Classes

Like a good application framework, PowerPlant provides a wide variety of utility classes. We will encounter some of these classes at various points throughout this manual. We discuss the rest in Appendix A, "PowerPlant Utilities."

Among the more important utility classes are:

- LString—for string manipulation
- UDebugging—for exceptions and signals while debugging
- UEnvironment—for gestalt-related operations
- UKeyFilters—for filtering keystrokes
- UModalDialogs—for managing a simple dialog that returns a single number or string
- UDrawingUtils—for determining drawing state
- UWindows—window management functions
- StProfileSection—for profiling parts of your code with the CodeWarrior Profiler

There are many, many more small classes for managing memory, preserving drawing state, and performing PowerPlant-related housekeeping such as registering classes.

In many of the utility classes, the member functions are static. You do not create objects based on those classes, you simply call the static member function to get the service provided.

According to PowerPlant naming conventions, classes that begin with the letters "St" are stack-based classes. Typically the class constructor does all the setup work, and the destructor does all the teardown work. Stack-based classes are frequently used in PowerPlant to preserve state information like the state of the pen, the current colors, the current grafPort, and so on.

Basic PowerPlant Resources

The final element of PowerPlant architecture is its resources. PowerPlant depends upon the presence of certain resources. Without them, a PowerPlant application simply won't work. They are:

- PPob Resource
- Mcmd Resource
- RidL Resource
- Txtr Resource

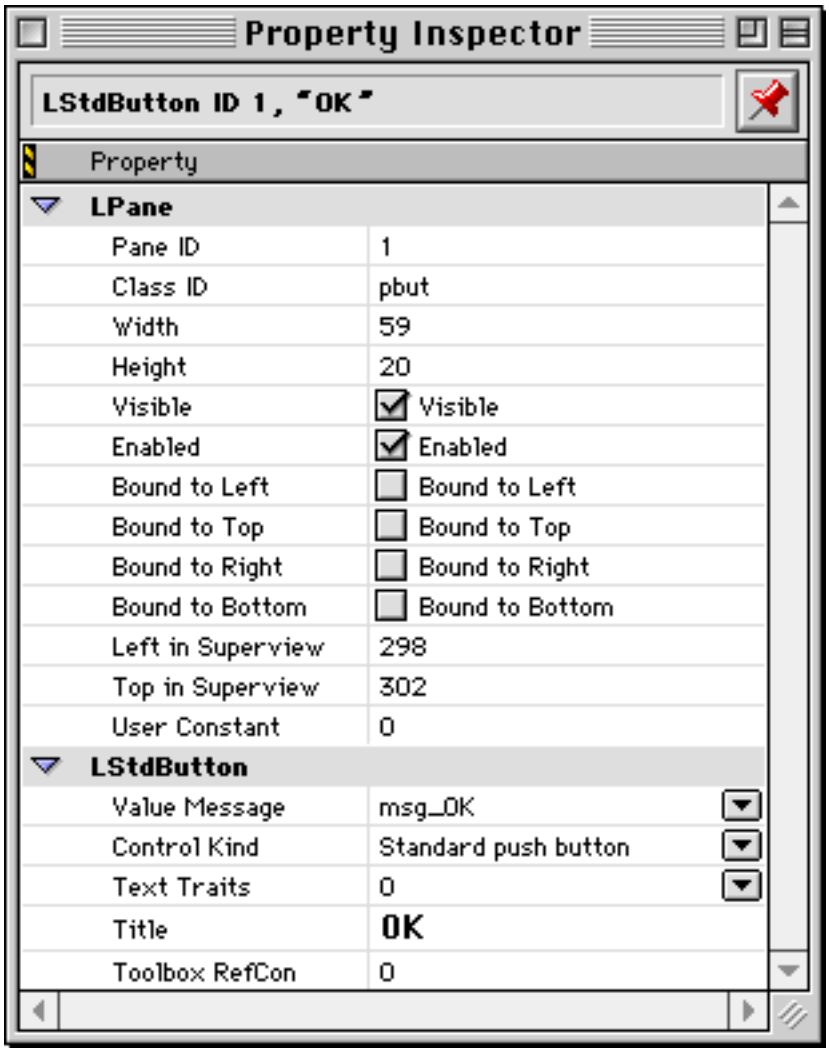
PPob Resource

A PPob resource describes the visual hierarchy: the various LView objects and panes, where they are located, and which LView objects contain which other LView objects and/or panes.

The easy way to create a PPob resource is with Constructor. Essentially, each class has a unique, four-character identifier, like the Finder's file type and creator codes for documents. When you issue a command to build a window based on a PPob resource, PowerPlant reads the data from the resource and recreates the necessary objects of the appropriate classes, based on the data you specified.

In Constructor you specify the nature of the pane or LView object, its position within the view hierarchy, and all the appropriate characteristics of the pane. For example, when you create a standard button you specify location, size, pane ID number, and a variety of other characteristics as shown in Figure 5.10.

Figure 5.10 Specifying a standard button in Constructor



We'll discuss each of these fields when you start building a PPob resource a little later. The point here is that you specify the complete visual appearance of a window using Constructor, and PowerPlant rebuilds the window from the PPob resource.

See also The *Constructor Manual* for details of Constructor's operation.

Mcmd Resource

PowerPlant uses specific command numbers for each menu item. In a classical Macintosh program, you determine what menu item the user has chosen by identifying which menu the user picked, and then what item number in the menu the user picked.

This approach works fine, but has one significant disadvantage. The code that dispatches menu choices is dependent upon the order of items in the menu. If you add, remove, or relocate an item in a menu, you must rewrite dispatch code to adjust for the change.

In PowerPlant, you assign each menu item a unique ID number, the *menu command number*. You store these numbers in Mcmd resources. There is one Mcmd resource for each menu, and each has a command number for each item in the corresponding menu.

Figure 5.11 shows the Constructor menu bar editor window. Each menu item has an associated command number. The command numbers are kept in an Mcmd resource. Constructor builds the Mcmd resource automatically. See the Constructor manual for details of how to build MBAR, MENU and Mcmd resources.

Figure 5.11 Menu bar, menus, and Mcmd resources

If you use Constructor to build your MENU resources, when you modify menus you don't have to change your source code at all. The commands stay with the menu item wherever you put it. At runtime, PowerPlant reads the menu and menu item selected, looks up the corresponding menu command number, and sends that number to your menu dispatch code for processing. Because the command number is constant, your code remains constant.

You can also build MENU and Mcmd resources with ResEdit, Resorcerer, or Rez. If you use these editors, you are responsible for making sure the MENU and Mcmd resources remain synchronized.

See also "Installing Resource Templates" on page 61 for information on installing Mcmd resource templates.

RidL Resource

A RidL resource is a list of one or more LControl pane IDs. A control is always a broadcaster but a broadcaster isn't necessarily a con-

trol. The RidL resource links a listener to controls and is a useful tool for messaging within a PowerPlant application.

In order for a listener to hear the message from a broadcaster, you must link the two of them. Suppose you have a dialog box (which inherits from LListener) that contains controls. If you want the dialog to "hear" the messages from the controls, you can use a RidL resource and the function `UReanimator::LinkListenerToControls()` to link it to each control in the dialog.

NOTE: There is another way to link a listener to an individual broadcaster. We'll discuss the topic of linking listeners with broadcasters in detail in "Broadcasting" on page 227.

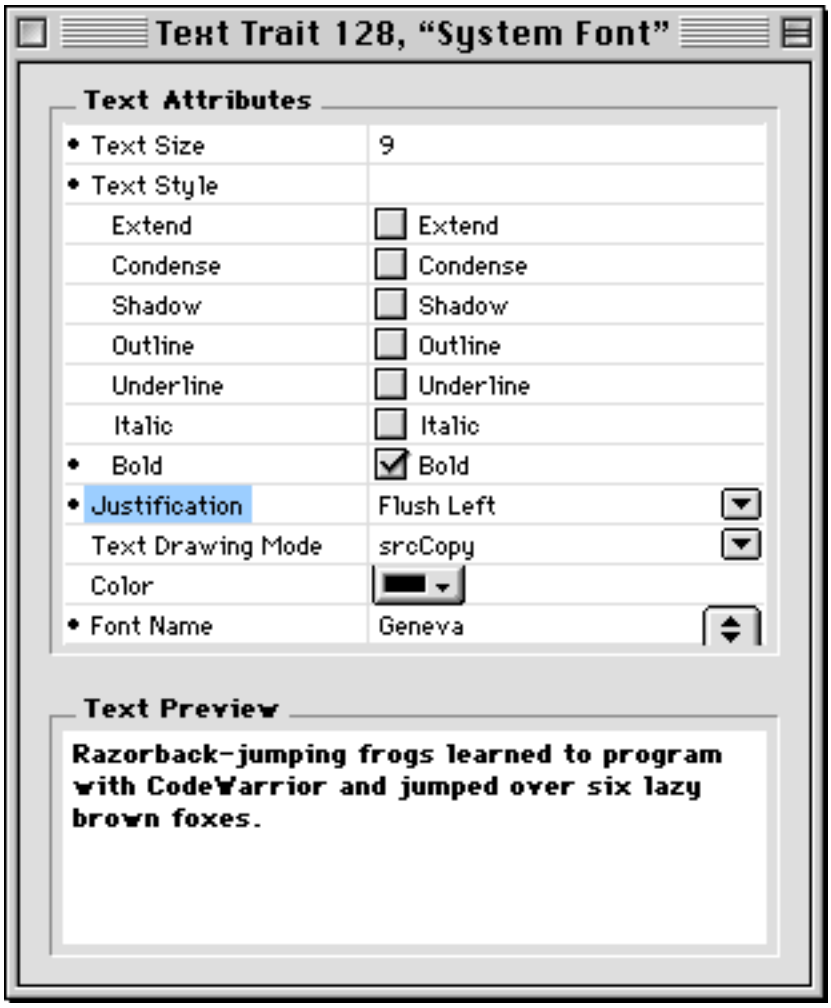
The Constructor view editor creates a RidL resource automatically for every window that contains controls. The RidL lists every nonzero control pane ID in the window. However, you cannot see or edit the RidL in Constructor.

There are other kinds of broadcasters besides controls, and they are not included in the automatic RidL resource. If you wish to make a custom RidL resource, or edit an existing resource, you must use ResEdit or Resorcerer.

Txtr Resource

The Txtr resource (text traits) describes the font, size, style, and alignment, color, and drawing mode for text. You can create and modify Txtr resources in Constructor. Figure 5.12 shows the text traits editor window for setting these traits.

Figure 5.12 Txtr Settings



When you create text-related panes in Constructor, you can specify a Txtr resource by ID number. The text in that pane will be drawn according to the values in the specified Txtr. You can switch the Txtr resource for a pane at runtime if you wish, by using the `SetTextTraitID()` function for the class.

The Txtr resource lets you encapsulate standard text-related settings into a single resource, which PowerPlant then uses to display text according to your wishes.

There are other PowerPlant resources, but the PPob, MBAR, MENU, Mcmd, RidL, and Txtr resources are the resources you see most frequently.

PowerPlant Development

Now that you have a clear picture of the architecture behind PowerPlant, you might be asking yourself, "What's it really like to write a PowerPlant application?"

If you have never written an object-based application before, you're going to find the process quite a bit different than what you're used to as a procedural programmer, and very rewarding.

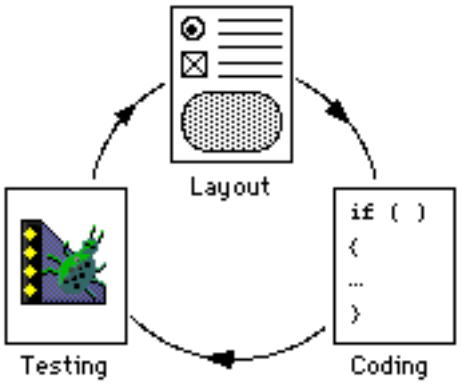
In this section we'll list the typical tasks you'll perform while developing a PowerPlant application. Of course, depending upon your personal style, and the needs of your project, you may perform the necessary tasks in a different order.

The development process breaks down into three segments:

- Layout
- Coding
- Testing

This is an iterative process where each segment provides feedback for the others.

Figure 5.13 The PowerPlant development process



This task list assumes you use Constructor to build an interface based on an LWindow object. Other objects may be the top-level LView object, including LDialogBox, LPrintout, LGrafPortView, or LView. You may not understand some of the items in the list of tasks presented here. However, keep them in the back of your mind. When you perform these tasks later in this manual, you'll recognize them.

Layout

Your first step (after designing your interface) is to do the initial layout work in Constructor. You performed most or all of these tasks in the code exercise in the Introduction to this manual.

- Make a new project file in Constructor, or use the starter resource from a new PowerPlant project and save it under a new name.
- Create a window.
- Design the layout by dragging panes from the Display Classes window to the layout view, and arranging the panes.
- Give each pane a unique ID, and note them for your later coding.
- For controls, start a sequence of message IDs and give each control a unique message ID.
- Use the Hierarchy window in Constructor to install objects inside of other LView objects, create any necessary radio button groups or tab groups

- Save and close your project file.

Of course, you won't get the entire layout done on the first run. You'll come back to these tasks again after you write some code and test your work.

TIP: Many PowerPlant developers like to have at least two resource files for a project. One contains the PPob, MBAR, MENU, Mcmd, Txtr, RidL, and WIND resources built by Constructor. The other contains the remaining resources. That way, you can have Constructor as the creator for your PowerPlant resources, and your favorite resource editor as the creator for your other resources. Include both resource files in your project. When you double-click the resource file, the right editor opens up.

See also The *Constructor Manual* for more information about Constructor.

Coding

Writing PowerPlant code is in principle the same as writing any kind of object-oriented code. However, you'll be writing code that takes advantage of PowerPlant's many outstanding features.

- Create a new PowerPlant project from the stationery.
- Add your PowerPlant project file (as noted above).
- Save the stationery starter source and header files with new names.
- Name and define your application class, and start implementing the required functions.
- Write the code to perform actions in response to commands and messages.
- Link listeners to broadcasters.
- Be sure that the MENU and Mcmd resources are correct, and enable any menu items that you will handle.
- Compile your file and fix compiler errors.

Testing

Of course, no project is complete without testing.

- Update the project, compiling all the PowerPlant library files
- Choose **Enable Debugger** from the Project menu.
- Make the project, and fix any linker errors.
- Choose **Run** from the **Project** menu to run your code with the Debugger.
- Check the appearance of the window.
- Use the menus and check the behavior of menu commands.
- Put breakpoints in the debugger windows wherever you want to trace the processing.
- Quit your application when you're done.

These tasks—layout, coding, testing—are not linear. You will perform all of these tasks more or less simultaneously at times. Running the project will allow you to discover ways of improving your interface, so you'll go back to Constructor and revise the resources.

You'll add new windows, perhaps floating palettes, dialog boxes, and other objects as necessary. Over time the interface will become more complex. However, building incrementally makes the process a lot more manageable.

Ultimately you will create your own pane classes, add them to the resources, and watch your application achieve its final form. It's an exciting process, and one that we are about to embark on.

In the next chapter we start looking at panes and LView objects (from now on simple "views") in detail. We'll examine the class hierarchy, and the typical view hierarchies you'll likely use. For now, let's look at where we've been.

See also The *Debugger User Guide* for information on debugging tools as well as the *PowerPlant Advanced Topics* chapter on the new PowerPlant Debugging Classes.

Summary

In this chapter you have learned many important concepts about the design philosophy behind PowerPlant, and how that design leads to a remarkably flexible and extremely powerful application framework for the Mac OS.

You have seen how that design extends itself directly into class implementation, function implementation, and even into the PowerPlant resources. Whenever possible, resources, functions, classes, and groups of classes are modular. You've met the principal classes and resources, and you know something about how they work together.

This is also the end of the background section of this manual. Taken together, the background chapters have given you a solid understanding of what an application framework is, why it is useful, how it works, and how PowerPlant in particular implements the features of a framework.

Now it's time to put that knowledge to work. In the next chapter you start serious work with an outstanding world-class development tool—PowerPlant.

6

Panes

This chapter and the next two chapters—on views and controls—form the Basic Building Blocks section of the book. These chapters deal with the visual objects you use in PowerPlant. Because an application framework is primarily a tool for creating a visual interface, panes, views, and controls are fundamental to PowerPlant.

This chapter discusses panes in general, and certain specific pane classes. Views and controls are also panes, but they have additional features that make them worthy of a separate discussion. Talking about views and controls in separate chapters also breaks the visual hierarchy in PowerPlant into more digestible bites.

The principle topics in this chapter are:

- What Is a Pane—including the different kinds of panes in PowerPlant.
- Pane Characteristics—a detailed look at the things that make a pane a pane.
- Working With Panes—how to make and use panes.
- Some Specific Panes—details on some pane classes.

After we complete this discussion, you'll create and manipulate real panes in this chapter's coding exercise.

What Is a Pane

In its most general sense, a pane represents a rectangular drawing area. In a more precise sense, a pane is usually a visible object such as a button or text box that appears in a view.

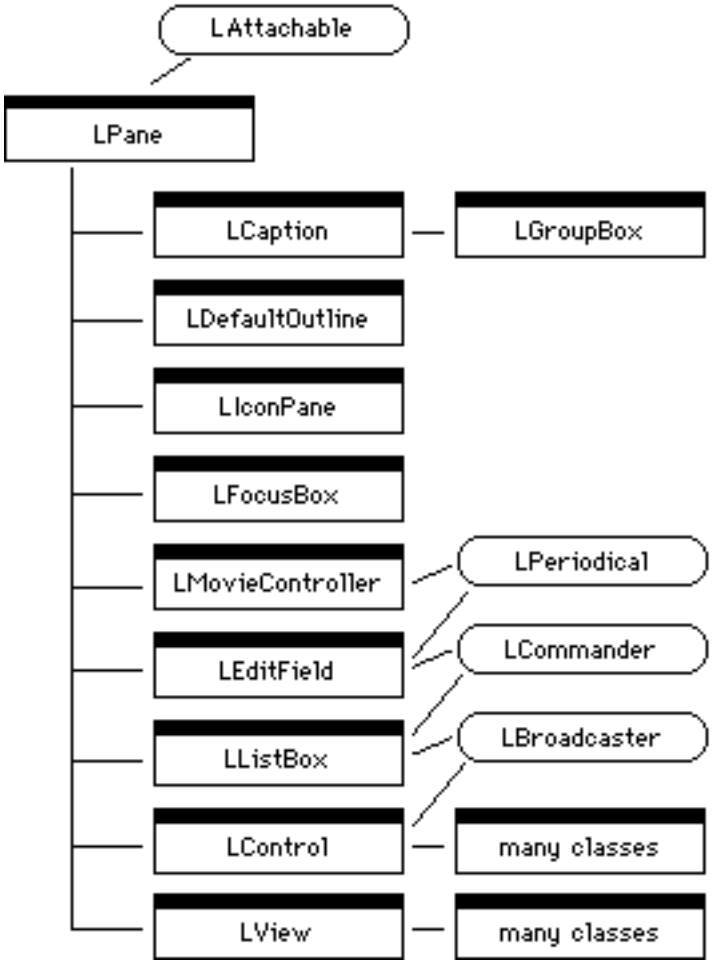
The fundamental pane class in PowerPlant is LPane. LPane describes a rectangular object that can display graphics. Some panes also respond to mouse clicks. You will rarely, if ever, create an ac-

tual LPane object. The LPane class serves to encapsulate a common interface for all the pane subclasses. Although LPane is not an abstract class, several important functions in the LPane class do nothing. They are overridden in LPane's descendants.

Now that you know what a pane is, let's talk about the different kinds of panes available in PowerPlant.

Figure 6.1 shows the class hierarchy for the pane classes.

Figure 6.1 Pane hierarchy



Notice that the LView class inherits from LPane. Chapter 7, "Views" discusses views in detail. The LControl class has several subclasses.

Chapter 8, "Controls and Messaging" covers controls. We'll talk about the other individual pane classes in "Some Specific Panes" on page 150.

TIP: For detailed information on any PowerPlant class, including a list of its ancestors, member functions, and data members, you can and should refer to the PowerPlant Revealed HTML reference.

In a PowerPlant application, you typically work with subclasses of LPane. LPane encapsulates a common interface for pane objects. As a result, all panes share certain common characteristics.

Pane Characteristics

In this section we talk about the various features of panes. We talk about how PowerPlant allows you to modify and manipulate those features in the next section, "Working With Panes" on page 136.

We're going to be careful here to not cause confusion with the terms "pane" and "view." The LView class inherits from LPane, and it has many subclasses. We discuss views extensively in the next chapter.

The principal distinction between a pane and a view is that a view can contain other panes. Therefore, we can divide pane classes into two groups: those that inherit from LView, and those that do not. Those panes that do not inherit from LView we will call "simple panes" because they cannot contain any other pane. If you look at the hierarchy diagram in Figure 6.1, the simple panes are all the classes in the diagram except LView and its descendants.

Although each of the view classes is a "pane" in the general sense, some of the characteristics we're about to discuss apply to simple panes—panes that are not also views.

Characteristics of Simple Panes

Panes that are not views comprise most of the real visual objects you draw on screen, including static text, editable text fields, buttons, check boxes, popup menus, icons, and so forth.

A simple pane is a leaf in the visual hierarchy. Every simple pane resides in a view of one sort or another. This view is called the pane's *superview*.

NOTE: Most views also reside in some other view. However, some views are at the top of the view hierarchy and have no superview (LWindow for example).

Simple panes use the coordinate system of their superview. The view is responsible for maintaining coordinates, as you'll see when we discuss views.

Finally, a simple pane cannot scroll its own contents. Views are responsible for scrolling. You can scroll panes inside a view, but you cannot scroll the contents of an individual simple pane.

See also "Coordinate Systems" on page 174.

Characteristics of All Panes

Everything discussed in this section applies to all panes, including LView and its descendants. Chapter 7, "Views" covers several additional features specific to views.

All panes have the following features:

- Pane ID
- Frame
- Frame binding
- Pane state
- Value and descriptor
- Mouse information

- Contents

To see how many of these characteristics are reflected in Constructor, see Figure 6.5 on page 137.

Pane ID

Each pane has an all-important *ID* number. Typically you assign the pane ID in Constructor when you define the pane's characteristics. The pane ID is a number of type `PaneIDT`, a 32-bit number. You may also specify the pane ID as a "text" ID—a sequence of four characters analogous to a resource type or file creator.

The LPane class has functions for managing the ID number, and finding a pane by ID. You will use `FindPaneByID()` regularly. Clearly each pane must have a unique ID or you're going to run into problems where the `FindPaneByID()` function returns a pointer to the wrong pane.

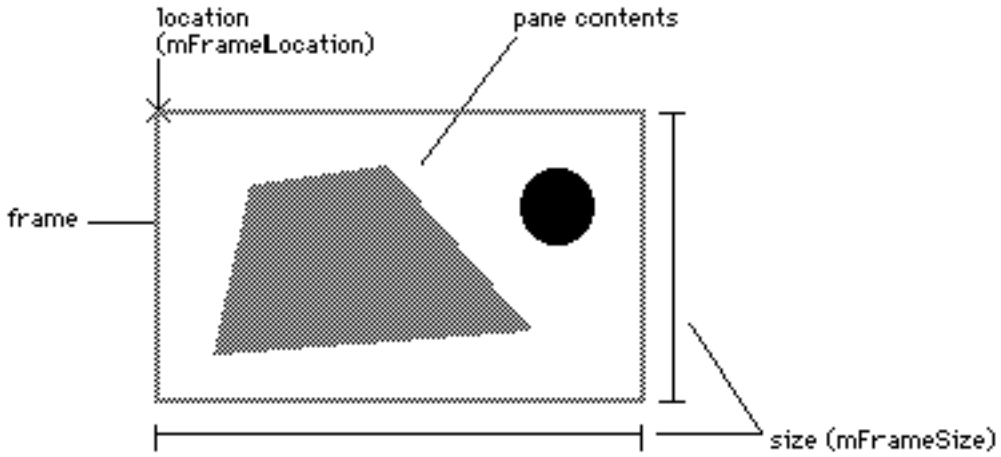
TIP: Actually, the limitation is that all panes in a single window for which you call `FindPaneByID()` must have unique ID numbers. You might have several panes with the same ID if you never look for them by ID number.

Frame

The **frame** is the rectangular area that the pane occupies. Like most rectangular areas in PowerPlant, the frame is specified by two structures: the **location** and the **size**. The location specifies the position of the top left corner of the rectangle in the superview's local coordinates. The size specifies the height and the width of the bounding rectangle.

See also "Coordinate Systems" on page 174.

Figure 6.2 Parts of a pane

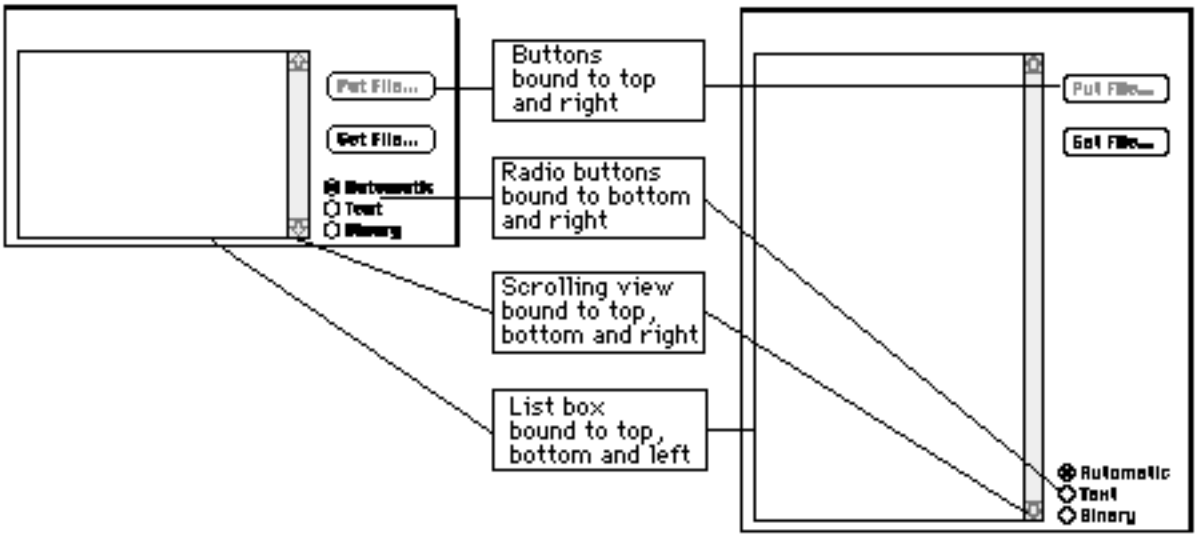


Frame binding

Each side of the frame has a binding option that specifies what happens to that edge when the pane's superview changes size. When an edge of a pane is bound, it is always the same distance from the corresponding edge of the superview. As a result, a pane may or may not change location or size in response to a change in the dimensions of its superview. Figure 6.3 illustrates the effect of binding on the size and location of panes.

Whether it is appropriate for a pane to change size or location depends upon the nature of the pane and the needs of your application. For example, a radio button should probably remain the same size no matter how the window grows or shrinks, but it may need to change position. A text object, on the other hand, may need to resize itself to fill its enclosing window.

Figure 6.3 Frame binding


Buttons bound to top and right
Radio buttons bound to bottom and right
Scrolling view bound to top, bottom and right
List box bound to top, bottom and left
Automatic
Text
Binary

Each pane has an `SBooleanRect` data structure that specifies frame binding. This structure is a series of four Boolean values, one for each of the four sides of the frame.

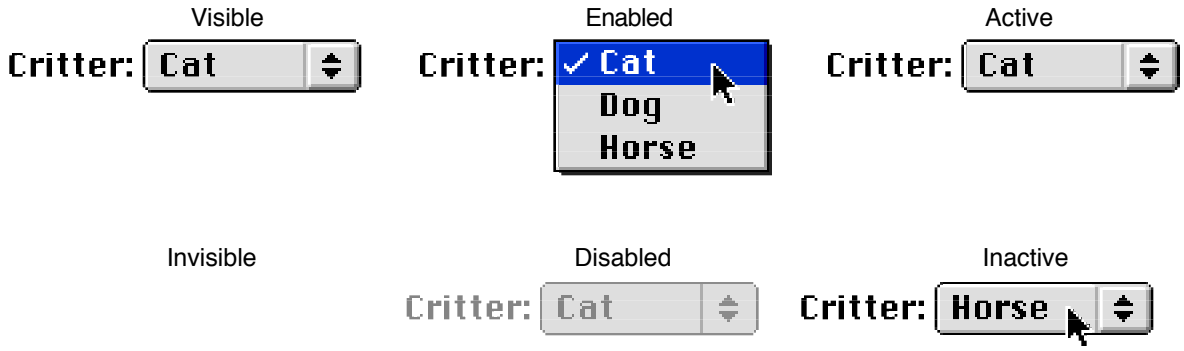
Pane state

Each pane has three states, as illustrated in Figure 6.4:

- Visible/hidden
- Enabled/disabled
- Active/inactive

Figure 6.4 uses a standard popup menu control to illustrate the effect of different states.

Figure 6.4 Pane states



The visible/hidden state controls whether you see the pane. The effect of visibility is obvious.

The enabled/disabled state determines whether the pane responds to clicks. For example, a click on a disabled popup menu results in no action.

The enabled/disabled state usually affects the pane's appearance, particularly if the pane is a control item. In the Mac human interface, disabled controls are dimmed as shown in Figure 6.4.

The active/inactive state refers to whether the window containing the pane is active or inactive. When you make a window inactive, that change propagates down through all the panes in the window. An inactive pane is not responsive to a click because it —and the window that contains it—are inactive.

The active/inactive state may or may not affect the visual appearance of the pane. For example, some controls looks the same in an inactive window. A scroll bar hides itself when inactive. A text pane may display a selection as an outline rather than highlighted when it is in an inactive window.

In a typical PowerPlant application, you may modify a pane's visible or enabled state if you need to do so. PowerPlant usually manages the active state for you.

When drawing a pane, consider whether the pane's state affects its appearance, and draw it accordingly.

Value and descriptor

Some panes have two other important features: a value and a descriptor. The *value* is a numerical representation of the contents of the pane.

While this is a feature available to all panes, only certain PowerPlant panes have a value, as summarized in Table 6.1. In each case, the value is an `Int32`—a 32-bit integer. The value feature is used for different purposes in different classes.

Table 6.1 Panes with values

Class	Purpose of value
LCaption	numerical equivalent of a text string
LControl (and descendants)	current value of the control
LEditField	numerical equivalent of a text string
LListBox	row number of first selected cell

For LCaption and LEditField, the value is useful if the text string represents an integer. In that case, you can use the value as the mathematical equivalent of the text string.

In addition to the value, some panes (Like LCaption and LEditField) also have a **descriptor**. This is a Pascal string. Like the pane's value, its purpose depends on the kind of pane, as summarized in Table 6.2

Table 6.2 Panes with descriptors

Class	Purpose of descriptor
LCaption	text in caption
LEditField	text in edit field
LListBox	text of first selected cell

Class	Purpose of descriptor
LStdControl (and descendants)	control title
LTextButton	button text
LWindow	title of window

TIP: The LSingleDoc class also has a Pascal-string descriptor. It is either the name of the associated file (if there is one) or the name of the associated window.

You have a great deal of flexibility in how you implement either the value or the descriptor in your own classes derived from LPane. However, when appropriate you should follow the convention that a pane's value represents a numeric quantity and its descriptor is the name of the pane or the textual representation of the value.

NOTE: The LPane class itself has no data member to store either value or descriptor. These are declared in the subclasses, when necessary. Some subclasses that use value and/or descriptor have data members to store this information. Other classes access data stored in Macintosh Toolbox structures. What LPane provides are the general functions for accessing the contents of the value or descriptor.

Panes also have a "generic" 32-bit data member, the `mUserCon`, that you can use for any purpose whatsoever. The purpose of `mUserCon` is analogous to the `refCon` field found in many Macintosh data structures.

Mouse information

The LPane class maintains several pieces of information related to mouse movements and actions as they affect the pane. This information is stored in data members as described in Table 6.3.

Table 6.3 Mouse-related information

Type	Data Member	Purpose
`LPane*`	`sLastPaneClicked`	pointer to the last pane clicked
`LPane*`	`sLastPaneMoused`	pointer to the last pane the mouse moved over
`Uint32`	`sWhenLastMouseUp`	time of last mouse up
`Uint32`	`sWhenLastMouseDown`	time of last mouse down (for timing double-clicks)
`Point`	`sWhereLastMouseDown`	location of last mouse down
`Int16`	`sClickCount`	number of recent clicks that are close in both time and space

If you need to examine the contents of any of these data members, use the accessors provided in the class. Each of these data members is static. In other words, these data members are pane globals. There is exactly one instance of each data member, and that instance is shared by all panes. As a result, you can always determine which was the last pane clicked, when it was clicked, if it was a double-click, and so forth.

Contents

A frame typically has something inside it. What is inside the pane depends entirely upon the nature of the specific pane. For example, the contents of a static text pane is a string of characters. The contents of a scrolling view may be a series of subpanes, text, or an image.

The contents are drawn by the specific pane's `DrawSelf()` function. Every subclass of LPane overrides this function to draw itself.

Working With Panes

LPane is a complex class with quite a few member functions. However, we can associate these functions into groups. Many of these groups are related to the characteristics we just studied. Grouping functions like this makes the purpose and use of the functions much easier to grasp. We're going to talk about

- Creating a Pane—using Constructor, and pane constructor functions.
- Drawing a Pane—drawing and updating panes.
- Managing Pane Characteristics—modifying or accessing the features of a pane such as ID number, frame, binding, and so forth.

NOTE: The LPane class also provides functions for managing coordinate systems. These are of primary importance in the view classes. We'll talk about these functions in "Managing Coordinate Transformations" on page 192.

Creating a Pane

You can create a pane using Constructor, or on the fly in your code. We talk about each method. Then we discuss what you do when you derive your own class from LPane or its descendants.

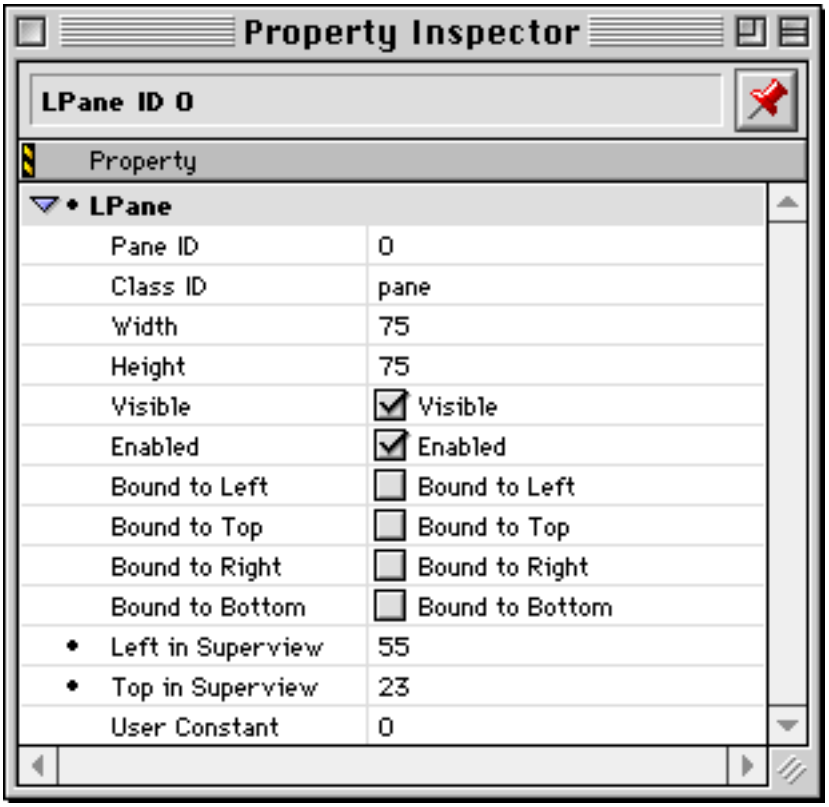
Using Constructor

You can use Constructor to define the characteristics of PowerPlant classes and your own derived classes. PowerPlant uses stream-based constructors to build entire containment hierarchies based on the 'PPob' data structure. You can edit a PPob in Constructor, Resorcerer, or Rez. The structure of the PPob resource is too complex for ResEdit to handle.

Creating a pane object in Constructor is simple. While in Constructor, you drag a pane object from the tool palette into a containing view. When you double click the object, an Info window opens so

you can set the characteristics for that object. The precise contents of the window will vary for each pane, but most of them have the fields shown in Figure 6.5.

Figure 6.5 Creating a pane in Constructor



The top left coordinate is relative to the pane's immediate superview. In Constructor, if you rearrange the view hierarchy to move a pane from one view to another, its position becomes relative to the top left corner of the new superview. We'll discuss views in the next chapter and revisit this point.

You're familiar with the Pane ID. Set it to a an appropriate value, (unique if you intend to access the pane by ID). You can use the User Constant for any purpose you see fit. It is a 32-bit value, and you can specify it as a number or a series of four characters (like a file's type or creator codes).

The Class ID field is a four-character code that PowerPlant uses to identify the appropriate routine for building the new pane.

If you are using PowerPlant classes, the class ID is set for you by Constructor with the correct value. When you derive your own classes, you must change the class ID to your own unique value. You must also register the class ID with PowerPlant before creating any objects of that class.

PowerPlant defines constants for the ID of each class that can be created from the 'PPob' resource. The value is an `enum` named `class_ID` specified within the declaration of each class.

For example, here's a snippet from the declaration of LCaption.

```
class LCaption : public LPane {
public:
  enum { class_ID = 'capt' };
```

This is a standard C++ technique used to define class-specific constants. You can access the class ID as `LCaption::class_ID`. You must provide a unique class ID in any derived pane class.

NOTE: PowerPlant reserves the set of all-lowercase class IDs for internal use. If you use at least one uppercase letter in your class ID, you will avoid a conflict with any and all PowerPlant classes, past, present, or future.

See also "Register PowerPlant Classes" on page 288.

Creating a pane on the fly

The typical approach used when creating a pane object on the fly is to define an `SPaneInfo` structure. This structure specifies the values required to build a generic pane. You then call the appropriate constructor. Depending upon the particular pane you are creating, you may need to provide additional information.

Listing 6.1 The SPaneInfo structure

```
struct   SPaneInfo {
  PaneIDT      paneID;
  Int16        width;
  Int16        height;
  Boolean      visible;
  Boolean      enabled;
  SBooleanRect bindings;
  Int32        left;
  Int32        top;
  Int32        userCon;
  LView*        superView;
};
```

Each pane class has specific constructors, one of which receives a pointer to the `SPaneInfo` structure. Most have additional parameters you must provide. Refer to *PowerPlant Revealed* for details on the various constructors and the parameters you must provide to successfully create a specific object on the fly.

After you have created a pane and installed it in a view, you should call `FinishCreate()`. This function ensures that the pane's state (visible/invisible, active/inactive, enabled/disabled) matches its superview. It also calls `FinishCreateSelf()`. The `FinishCreateSelf()` function gives you the opportunity to provide "finishing touches" when creating a pane or view, because there may be times when you can't fully initialize a pane in its constructor.

For example, for performance reasons you may want a view to maintain pointers to some of its subpanes. (This saves the overhead of repeatedly calling `FindPaneByID()` when the view wants to access a subpane.) You cannot initialize the view's list of subpanes during view construction, because subpanes are created after the superview. However, you can override `FinishCreateSelf()` to create the list after the subpanes are built.

There are additional member functions in the LPane class that you may use when creating a pane on the fly, if you don't use the `SPaneInfo` structure.

You may use `PutInside()` to make the pane a subpane of a view. To remove a pane from a superview, call `PutInside()` and pass nil as the new superview.

`PlaceInSuperFrameAt()` places the pane at a location relative to the superview's frame and `PlaceInSuperImageAt()` places the pane at a location in the superview's image. The distinction between a view's frame and its image is discussed in the "Views" chapter, in the section "Image" on page 172.

See also "Managing Pane Characteristics" on page 145 for information on setting individual features of a pane.

Deriving your own panes

When you derive a class from LPane or one of its descendants, you typically define a class creator function and several constructors: a default constructor, a constructor that receives an `SPaneInfo` structure, a copy constructor, and a constructor to build the pane from a stream. The class creator function and the stream constructor are worthy of special attention.

See also "Creating a pane on the fly" on page 138 for more on the SPaneInfo structure.

Class creator function

Older PowerPlant classes use a creator function when creating a pane-based object. These creator functions are static. The function receives a pointer to an LStream (the stream that contains the data from which to create the object), and returns a pointer to the new object. The prototype for a CCustomPane creator function is listed here as an example.

```
static CCustomPane* CreatePaneStream(LStream *inStream);
```

This method is no longer used, but may still be encountered in older classes. Class creator functions can be safely removed from any class that uses them.

When you derive a pane class, you must provide a stream constructor and a class_ID.

See also "Register PowerPlant Classes" on page 288.

Stream constructor

A stream constructor receives a pointer to an LStream object, reads data from the stream, and builds the object based on that data.

Here's the code for the LPane stream constructor.

Listing 6.2 The LPane stream constructor

```
LPane::LPane(LStream* inStream)
{
  SPaneInfo thePaneInfo;
  inStream->ReadData(&thePaneInfo, sizeof(SPaneInfo));
  InitPane(thePaneInfo);
}
```

Chapter 13, "File I/O" discusses LStream in more detail.

This code tells the stream to read in a certain amount of data and put it in an `SPaneInfo` structure. It then calls `InitPane()` to initialize the object based on that information.

When you derive your own pane classes, you must provide a stream constructor. If your pane does not need to read any data from the stream, you *still need* to have an LStream constructor, but your derived stream constructor can simply call the base class's stream constructor.

For example, assume you derived a class from LIconPane, and it needed no additional data. Your stream constructor might look like this.

Listing 6.3 Stream constructor for a hypothetical icon pane class

```
CMyIconPane::CMyIconPane(LStream* inStream)
  : LIconPane(inStream)
{
}
```

If you need to create custom panes that have additional data, you can do so in Constructor. See the *Constructor for PowerPlant Guide* for details. In this case, your stream constructor would read the additional data and initialize the object based on that data.

Overriding LPane functions

Of course, you may override whatever functions are necessary in your own pane class. The functions you are likely to override include:

Table 6.4 Commonly overridden pane functions

Function	Purpose
`ClickSelf()`	respond to a mouse click
`DrawSelf()`	draw the pane contents
`GetValue()`	get the pane's numerical value
`SetValue()`	set the pane's numerical value
`GetDescriptor()`	get the pane's descriptor string
`SetDescriptor()`	set the pane's descriptor string

Of course, you would only override the value and descriptor accessors if your pane class used those features.

NOTE: Views and controls have additional functions specific to those types of classes that you would typically override. See "Creating a View" on page 180, and "Creating a Control" on page 221.

Drawing a Pane

Like most Macintosh applications, PowerPlant draws the contents of a pane when an update event occurs. When your application receives an update event for a window, PowerPlant's default behavior calls the window's `UpdatePort()` member function, which in turn calls the window's `Draw()` member function. `Draw()` sets up the

coordinate system (described in more detail in "Drawing a View" on page 187), calls the window's `DrawSelf()` function, and then calls `Draw()` for each subpane in the window.

The pane's `Draw()` function does the necessary setup work. The default `LPane::Draw()` function prepares for drawing the pane by calling the local `LView::FocusDraw()`. Panes rely on the superview to set the focus and manage coordinate transformations.

TIP: The preferred way to draw a pane is to call the `Draw()` function. You should never call `DrawSelf()` directly. If you do any drawing that does not go through the `Draw()` function, you must call the view's `FocusDraw()` directly to ensure that the port and coordinate system are set up correctly.

After setting the focus (and performing some other housekeeping details), the pane's `Draw()` function then calls `DrawSelf()`. All classes derived from LPane must override `DrawSelf()` to draw the contents of the pane.

To draw the contents of a pane or a view, you use standard Macintosh drawing routines as you would for any other Macintosh program. PowerPlant does not replace QuickDraw.

For example, Listing 6.4 shows the `DrawSelf()` function for LStdControl—the class that represents standard Macintosh controls. (The actual code is more elaborate, but this gives you the idea.)

Listing 6.4 LStdControl::DrawSelf()

```
void LStdControl::DrawSelf()
{
  // mMacControlH is a data member of LStdControl
  // that contains a Macintosh control handle

  ::Draw1Control(mMacControlH);
}
```

For a pane that draws an X from one corner of the pane to the other, the `DrawSelf()` function might look like this:

Listing 6.5 DrawSelf() for a derived pane

```
void MyPane::DrawSelf()
{
  Rect   frameRect;

  // CalcLocalFrameRect returns the pane's frame
  // as a QuickDraw rectangle in local coordinates
  CalcLocalFrameRect(frameRect);
  ::MoveTo(frameRect.left, frameRect.top);
  ::LineTo(frameRect.right, frameRect.bottom);
  ::MoveTo(frameRect.right, frameRect.top);
  ::LineTo(frameRect.left, frameRect.bottom);
}
```

Validating and invalidating drawing areas

On occasion, you may wish to force an update event or prevent one from happening. The LPane class provides the functions listed in Table 6.5 to assist you.

Table 6.5 Validating and invalidating areas

Function	Purpose
`Refresh()`	invalidate the entire pane
`DontRefresh()`	validate the entire pane
`InvalPortRect()`	invalidate the rectangle specified
`ValidPortRect()`	validate the rectangle specified
`InvalPortRgn()`	invalidate the region provided
`ValidPortRgn()`	validate the region specified

The rectangle or region specified should be in port coordinates.

WARNING! You should use these routines rather than the corresponding Toolbox calls `InvalRect()`, `InvalRgn()`, `ValidRect()`, and `ValidRgn()`. For one thing, the PowerPlant calls handle coordinate transformations correctly. In addition, the Toolbox calls require that the current GrafPort be a window. However, a pane could be in another kind of GrafPort, such as a printer port or a GWorld. If the pane is not in a window, calling one of these Toolbox routines will cause a crash (when the Toolbox tries to access a nonexistent update region).

Additional drawing considerations

If your pane draws or behaves differently when its state changes, override the functions `ActivateSelf()`, `DeactivateSelf()`, `EnableSelf()`, and `DisableSelf()`.

Managing Pane Characteristics

As you know, panes have many features. PowerPlant lets you adjust those features freely.

View hierarchy

Panes reside in views. Typically you set the view hierarchy in a PPob resource using Constructor. However, you can modify the view hierarchy at runtime if you wish. This lets you create panes on the fly and install them in existing views.

To get a pane's current superview, use `GetSuperView()`. To put a pane inside a view, use `PutInside()`.

Pane ID

Every pane has a unique ID that you can retrieve with the function `GetPaneID()`. You can set this value with `SetPaneID()`. You won't normally need to use the setter function. The pane ID is typically set in Constructor, or by using the `SPaneInfo` constructor function appropriate for the pane you are building on the fly.

The more common occurrence is that you want a pointer to a pane when you already know the ID number. You usually know the ID number because you assigned it at some point. To get the pointer, you call `FindPaneByID()`. You will use this function often. It searches the current view hierarchy and returns a pointer to the specified pane. This is analogous to the Mac Toolbox routine `GetDialogItem()` used to retrieve a pointer to a dialog item.

TIP: You can also retrieve a pane hit by a mouse click with `FindSubPaneHitBy()`. However, this function is only useful in views (which have subpanes).

Frame

Recall that the pane's frame is described by two structures, the location and size. Table 6.6 lists some functions you may use for managing frame characteristics.

Table 6.6 Some frame management functions for panes

Function	Purpose
`GetFrameSize()`	returns current size
`GetFrameLocation()`	returns current location
`ResizeFrameTo()`	set new size to absolute value
`ResizeFrameBy()`	set new size to relative value
`MoveBy()`	relocate frame by a relative value
`CalcPortFrameRect()`	get the frame in port coordinates
`CalcLocalFrameRect()`	get the frame in local coordinates
`PlaceInSuperFrameAt()`	position frame in superview frame at absolute coordinates
`PlaceInSuperImageAt()`	position frame in superview image at absolute coordinates

Frame binding

The frame binding is usually specified in a PPob resource using Constructor, or in the `SPaneInfo` record if you build the pane on the fly.

Use `GetFrameBinding()` to retrieve the current settings, and `SetFrameBinding()` to change the current settings.

Contents

If you derive a class from LPane or its descendants, you are responsible for rendering the appearance. See "Drawing a Pane" on page 142 for full details. You can study PowerPlant's own panes to see how they draw themselves.

Value and descriptor

Use `GetValue()` and `SetValue()` to access the value data member. Use `GetDescriptor()` and `SetDescriptor()` to access the descriptor data member.

Remember that all pane classes have the value and descriptor accessors, but only a few PowerPlant classes actually use them. In LPane, the accessors do nothing.

NOTE: If you are using PowerPlant's debugging features, these accessors throw a signal to alert you to inappropriate use.

Use `GetUserCon()` and `SetUserCon()` to access the `mUserCon` data member.

State

Table 6.7 lists the pane functions used to manipulate the pane's state. You can query the pane to determine current state, and set the state to an appropriate value.

Table 6.7 Pane state-related functions

Function	Purpose
`IsVisible()`	returns true if pane is visible
`Show()`	make a pane visible
`Hide()`	make a pane invisible
`IsActive()`	returns true if pane is active
`Activate()`	make a pane active
`Deactivate()`	make a pane inactive
`IsEnabled()`	returns true if pane is enabled
`Enable()`	make a pane enabled
`Disable()`	make a pane disabled

TIP: These functions modify the pane's state, not its behavior or appearance. Override `ActivateSelf()`, `DeactivateSelf()`, `EnableSelf()` and `DisableSelf()` for changing appearance or modifying behavior as state changes.

Adjusting the cursor

Mouse information in panes is typically maintained for you automatically by PowerPlant. You may wish to perform two mouse-related tasks: adjusting the cursor and identifying whether a click hits a particular pane.

If your pane uses its own cursor, override `AdjustCursorSelf()`.

You should also be aware of the `MouseEnter()`, `MouseWithin()`, and `MouseLeave()` functions. These are empty member functions defined in the LPane class. PowerPlant does not use these functions at all. They allow you to implement your own mouse tracking. For example, you might create a pane that also inherits from LPeriodical. The `SpendTime()` function would be called regularly, and would determine if the mouse was entering, within, or leaving a

pane. The `SpendTime()` function would then call `MouseEnter()`, `MouseWithin()`, or `MouseLeave()` as appropriate.

You can also manage cursor adjustment using attachments. Create a subclass of LAttachment that responds to `msg_AdjustCursor`, and attach it to the pane. Your attachment could be a highly reusable bit of code that you could connect to almost any pane when cursor adjustment was important.

See also "Periodicals" on page 506 and "Attachments" on page 512.

Hit testing

PowerPlant manages most hit testing for you. There will be times when you'll want to test whether the mouse is in a pane, and whether a click is in a pane. You'll also respond to clicks.

The LPane class has the functions listed in Table 6.8.

Table 6.8 Hit testing in panes

Function	Purpose
`Contains()`	returns true if a point is within a pane, regardless of pane state
`IsHitBy()`	returns true if a point is within a pane, and the pane is enabled
`GetLastPaneClicked()`	returns a pointer to the last pane clicked
`Click()`	performs click-related housekeeping, calls `ClickSelf()`
`ClickSelf()`	respond to a click
`FindSubPaneHitBy()`	for views, find subpane of this pane that contains the point

Function	Purpose
`FindDeepSubPane Containing()`	for views, search through subpanes for deepest pane containing the point
`FindShallowSubPane Containing()`	for views, search through subpanes for shallowest pane containing the point

Most hit testing is provided for you automatically by PowerPlant. In a simple application, the only function in Table 6.8 that you'll override in your own classes is `ClickSelf()`.

The three functions related to finding subpanes are useful for LView and its descendants. These are the only classes of panes that can contain subpanes. In the LPane class these are empty functions.

Some Specific Panes

Now that you have absorbed all that knowledge about panes in general, let's take a quick look at some specific pane classes and the features that are unique to them. Remember, we'll be discussing views and controls in subsequent chapters.

In this section we'll talk about every pane class that is neither a view nor a control. The classes covered are:

- LCaption—display static text
- LGroupBox—display bounds of a group of items
- LDefaultOutline—outline the default button
- LIconPane—display an icon
- LFocusBox—display a black box around a frame
- LMovieController—display a QuickTime movie controller
- LEditField—display an editable text box
- LListBox—display a Macintosh List Manager list

LCaption

LCaption displays text. This class uses a text traits resource to specify characteristics such as font, size, style, color, and justification. LCaption uses the UTextDrawing class to draw text.

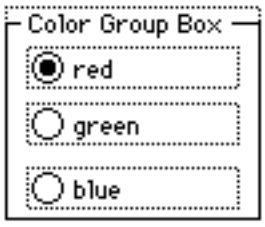
You can set the text and the text traits resource in Constructor, or you can modify these characteristics at runtime.

You may encounter a drawing problem if you change the contents of the caption at runtime. `LCaption::DrawSelf()` uses `UTextDrawing::DrawWithJustification()`. This does not erase the previous contents of the caption. The best way to erase the contents is to attach an LEraseAttachment object to the caption.

See also "Attachments" on page 512.

LGroupBox

This class derives from LCaption. The text is the title of the group. The object draws a box around the confines of the group.



Note that the top of the object's frame does not coincide with the top line drawn for the group box.

The other panes that are visually within the group box do not "belong" to the group box in any hierarchy. They simply overlap visually. The group box is a visual decoration. It does not control or affect the panes within the box in any way.

LDefaultOutline

The primary use for this class is to draw an outline around the default button in a dialog. You will typically not create an object of this class yourself. You cannot create one in Constructor. PowerPlant makes one for you when you create a dialog and specify a default button.

LIconPane

LIconPane draws a single icon from an icon family. It stores the ID of an icon family and draws it on the screen. Because it uses the Mac OS icon-handling routines, an LIconPane object draws the member of the icon family that best fits the color status and bit depth of the current device.

LFocusBox

An LFocusBox object outlines a pane to indicate that the pane is the current focus for keystrokes. This class is used internally by PowerPlant in conjunction with LListBox to highlight entries in the list. Like LDefaultOutline, you will typically not create an object of this class directly. You cannot create one in Constructor.

LMovieController

This is a wrapper class that creates, draws, and disposes of a standard QuickTime movie controller. You would typically use this in conjunction with UQuickTime if your application supports QuickTime movies.

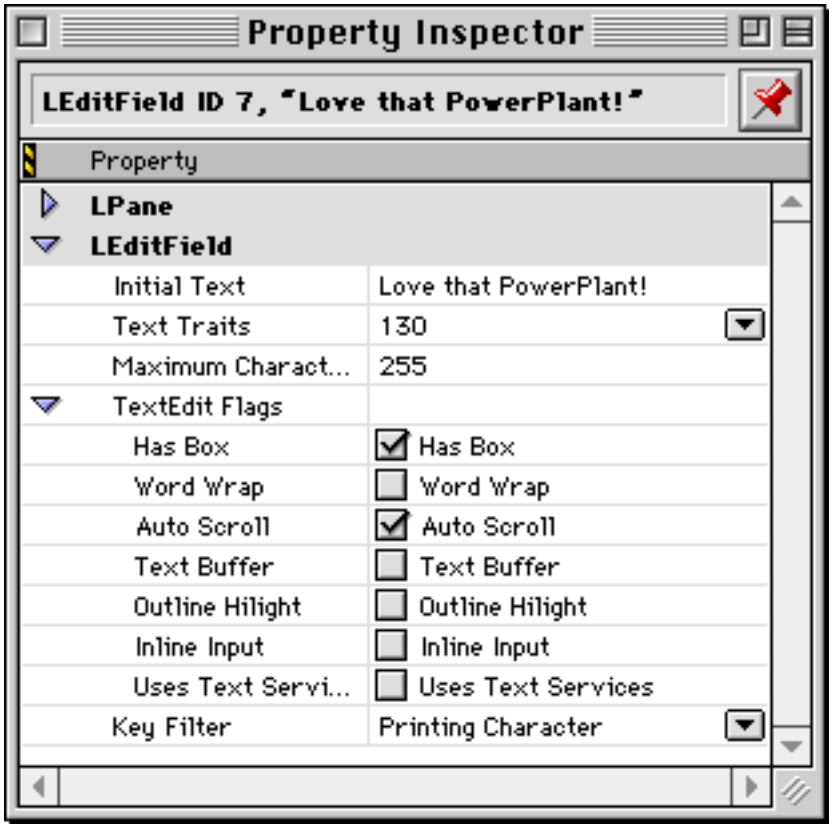
The LMovieController class inherits from LPeriodical, so it can receive and process every event retrieved by the event loop.

LEditField

LEditField uses single-style TextEdit to implement an editable text field, such as those in standard Macintosh dialog boxes. PowerPlant also handles undo and redo for most text-related actions. This class derives from LPane, LCommander, and LPeriodical.

This object does not have a scrollbar. You can set the object to "autoscroll," meaning that the displayed text will scroll as the text cursor moves through the text. Figure 6.6 shows some of the options you can set in Constructor. The initial text and text traits can also be set using Constructor.

Figure 6.6 Some LEditField options



Notice that PowerPlant allows you to attach a key filter to the LEditField. There are several default key filters available. Look up the UKeyFilters class in *PowerPlant Revealed* for more information. PowerPlant also implements Undo and Redo for most standard actions.

If you have two or more edit fields in a view, human interface guidelines suggest that typing the Tab key should advance the text entry cursor from one field to the next.

You can implement this behavior very easily by using a helper class named LTabGroup. In Constructor, you choose the **Make Tab Group** item from the **Arrange** menu. This adds the LTabGroup object to the PPob resource.

The LTabGroup object is faceless—you cannot see it. It is not a pane. LTabGroup inherits from LCommander. It will receive the Tab key-

press and automatically shift the target object to the next editable text object in the view. LTabGroup also supports Shift-Tab for moving to the previous editable text object.

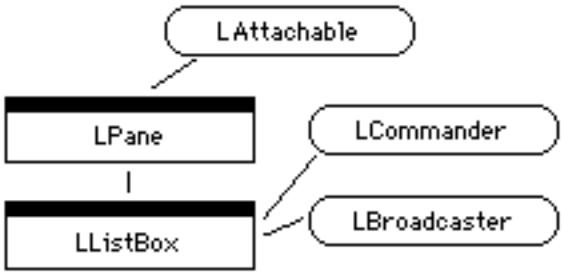
TIP: Some key filters handle Tab keypresses before the tab group gets them. You may need to create your own filter to ensure that Tab keypresses are passed to an LTabGroup object.

See also "LTextEditView" on page 196.

LListBox

LListBox is a wrapper class for the Mac OS List Manager. With some work, this class allows you to create a two-dimensional table of cells that scroll, respond to keystrokes, and so forth. Figure 6.7 shows the inheritance hierarchy that leads to LListBox.

Figure 6.7 LListBox hierarchy



If you use Constructor to create an LListBox object, you can only create a list with one column. To add columns (or rows) on the fly, call `GetMacListH()` to get the Toolbox ListHandle. Then use the List Manager—calling `::LAddRow()` and/or `::LAddColumn()`.

The default LDEF resource (list definition) used by the Toolbox draws text in each cell. If you want to display other types of data, you must provide your own LDEF resource.

TIP: The LTable class provides basically the same functionality as LListBox with greater flexibility and better performance. See

"LTable" on page 197. The LTableView class is even more powerful. LTableView is discussed in PowerPlant Advanced Topics.

The `ClickSelf()` function in LListBox responds to double-clicks. All clicks are passed to the List Manager for processing by the Toolbox `LClick()` routine. If that call returns `true`, the click in the cell was a double-click. LListBox broadcasts a message to that effect so that any listeners can respond.

If you want listeners to be aware of single clicks, you must derive a new class from LListBox and override the `ClickSelf()` function. Listing 6.6 shows one way to do that.

Listing 6.6 Broadcasting single clicks from a list box

```
void
CMyListBox::ClickSelf (const SMouseDownEvent &inMouseDown)
{
  SwitchTarget(this);
  FocusDraw();

  if (::LClick(inMouseDown.whereLocal,
        inMouseDown.macEvent.modifiers,mMacListH))
  {
    BroadcastMessage(mDoubleClickMessage, this);
  } else {
  // msg_SingleClick is a const that you define
    BroadcastMessage(msg_SingleClick, this);
  }
}
```

Summary

Panes are the fundamental visual objects in PowerPlant. There are many kinds of panes, including views, controls, LCaption, LGroupBox, LIconPane, LEditField, LMovieController, and others.

Panes have a variety of characteristics including position in a view hierarchy, ID number, frame, frame binding, contents, value, descriptor, state, and mouse information.

You typically create a pane with Constructor. You can build panes on the fly in code, using the `SPaneInfo` structure and passing any other necessary data to the constructor function. You can get, set, or otherwise manipulate every characteristic of a pane at runtime.

You have just absorbed a tremendous amount of information about panes. As we said before, the LPane class and its derivatives are fundamental to all the visual objects in PowerPlant. The LPane class is correspondingly complex. However, you now have a really solid understanding of panes and their offspring. That will make understanding views and controls much easier in the next chapters.

Let's put all this knowledge to work and see how to create and use panes in code.

Code Exercise

Because this is the first code exercise, let's be explicit about how these exercises are designed, and about some assumptions we're going to make. You can use these exercises in several ways. They are designed to be as flexible as possible to accommodate your individual learning style.

Learning Paths

Each exercise has a "start code" and a "solution code." The exercise itself is a series of steps in which you add code to one or more files in the start code project. In the process you learn step by step how to implement some feature in PowerPlant. In this chapter, for example, you learn how to work with panes.

WARNING! Do not attempt to build and run the start code project without performing the steps. Critical code is missing, and the project will either not build, or crash when it runs.

The solution code represents the complete project after you have followed all the steps. It contains all the code that the exercise steps ask you to add, and all the resources you build in Constructor.

Each exercise is a series of steps. The step title specifies a particular task you should accomplish. The code locator specifies the precise file and function involved in the step. The step instructions explain what you are supposed to accomplish in the step. Each step also includes the code you must write. We frequently include some existing code as well, so you can locate the precise spot where you should be adding new code. Existing code is in italic.

This design gives you at least three strategies you can use as you perform an exercise.

First, there is the "tutorial" strategy. With this approach, you follow the steps precisely and copy the code exactly as provided. This technique guarantees that your final project will work. You will be "following along in the book" as you implement the task at hand. You will learn a lot about PowerPlant from doing it as experts do.

Second, there is the "guide" strategy. In this approach, you read the step instructions, ignore the code, and solve the problem yourself. You use the steps as a guide to your own work. This technique is more suited for the adventurous programmer. If you pursue this tack, you are likely to make mistakes and your project may not work right the first time. As you eliminate the bugs, you'll learn a lot about PowerPlant by finding your own way.

Third, there is the "example" strategy. In this approach, you don't write any code. You use the steps in the exercise as an explanation of example code. In this case, use the files in the solution code project. As you read the steps, study the code. This approach is best suited to someone who wants a theoretical understanding of the inner workings of PowerPlant, or someone who wants a little guidance but is eager to apply the principles directly in their own project.

Basic Assumptions

For each chapter, there is a folder titled "Chap nn Start Code" where "nn" is the chapter number. These folders are located in the "PP Book Code" folder. We assume that you have located the appropriate start code and copied it to your hard drive.

Inside each start code there is a CodeWarrior project file. There are project files for both 68K and PowerPC code. We assume that you have launched CodeWarrior and opened the project file.

Immediately after the step title, you will usually see a code locator:

```
function name()                                    File.cp
```

This identifies the file and function you work on in the step. We assume you have opened the file and located the function.

We're going to assume that you'd rather write code than build PPob resources. As a result, we're going to provide most of the PPob resources required to perform these exercises. You should keep in mind, however, that this is a gift. In your own work you must build PPob resources, usually from scratch.

Finally, a few words about the project files in these exercises. These projects are not derived from the standard PowerPlant stationery files. We wanted to make sure that all of the necessary files were included in the project, so you can concentrate on learning PowerPlant, not adding and removing files in CodeWarrior.

We'll explore some of the subtle differences in the code exercise for Chapter 9. However, there are some apparent differences worthy of note here.

First, the `PPStarterResources.rsrc` file has been replaced with two files, `appname.rsrc` and `appname.ppob`, where "appname" is the name of the application in that project. As we mentioned in the code exercise in the introduction, this makes it easy for you to open the file in the correct resource manager application. Some of the resources in `PPStarterResources.rsrc` have been omitted because they are unnecessary in these projects.

We have also added two resource files, `PP AppleEvents.rsrc` and `ColorAlertIcons.rsrc`. The first includes the 'aedt' resource for PowerPlant. (The 'aedt' resource is part of the `PPStarterResources.rsrc` file.) The second replaces the black and white alert icons with color icons. Appendix B, "Resource Notes" discusses the PowerPlant resource files and their contents. For more information, see "ColorAlertIcons.rsrc" on page 576 and "PP AppleEvents.rsrc" on page 576.

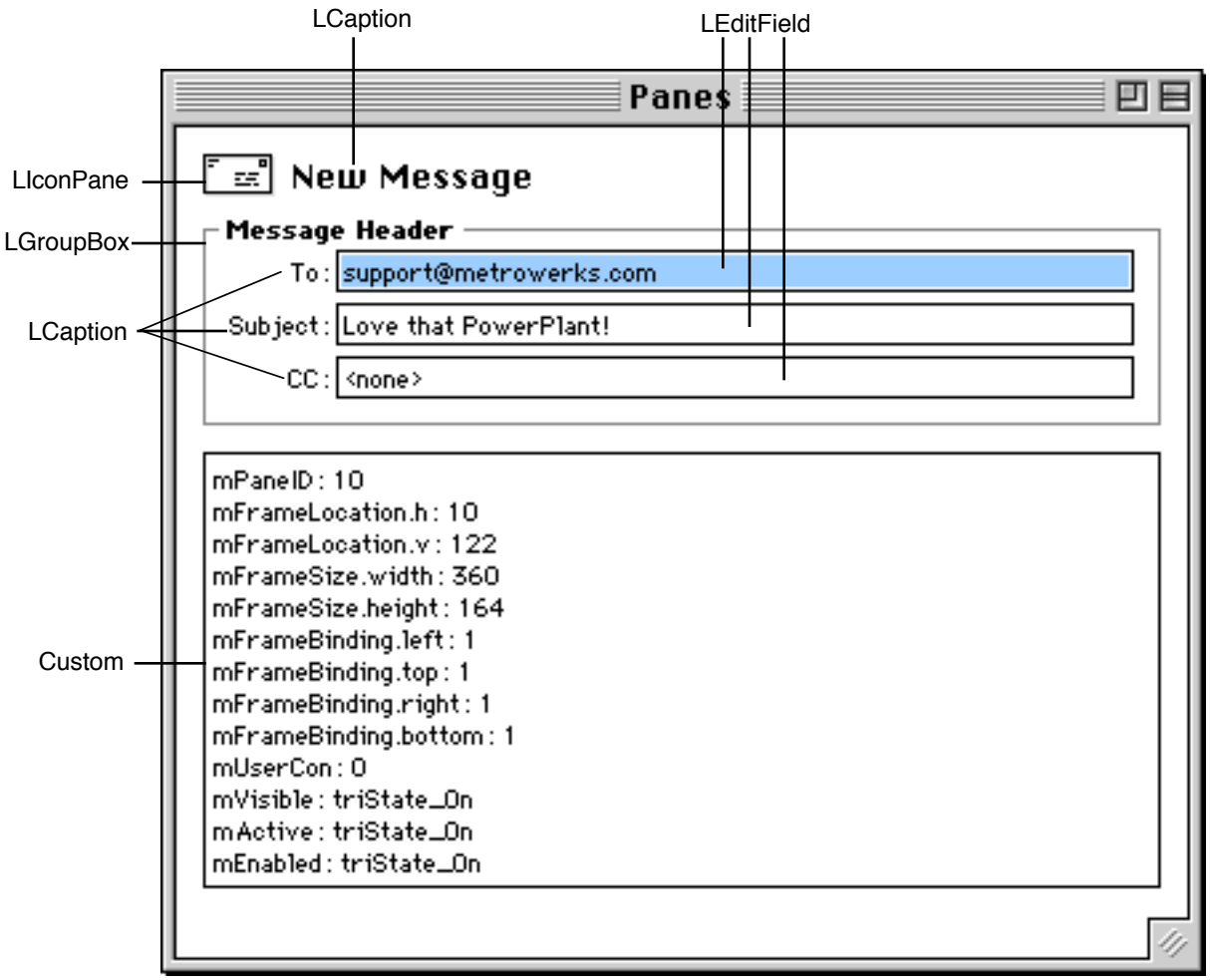
There are a few more changes behind the scenes with the project prefix, included files, and so on. We'll point those out in Chapter 9 when you work on the background details in a PowerPlant application like setup, debugging, and memory management.

Now that we have the ground rules established, we can begin our adventure.

The Interface

In this project you build an application titled "Panes." The final product looks like Figure 6.8.

Figure 6.8 The Panes window



There are ten panes in this window, as noted in the illustration. If you examine the PPob resource with Constructor, you learn the following about these panes.

The LIconPane is bound to the top left of the window. It is not enabled, so it won't respond to clicks.

All LCaption objects are bound to the top left of the window. None is enabled. The "New Message" caption uses a text traits resource of zero—the system font. The others use a text traits resource ID 131—Geneva 9 point, flush right.

The LGroupBox is bound to the top, left, and right of the window. As the window changes width, it will too. It is not enabled. It uses a text traits resource ID 132—Geneva 9 point, flush left, bold.

The LEditField panes are all bound to the top, left, and right of the window. As the window changes width, they will too. They are enabled, so they can respond to clicks and keystrokes. Each uses a key filter to limit input to "legal" characters. Each uses the text traits resource ID 130, Geneva 9 point, flush left.

With a series of editable text fields, the Mac OS human interface guidelines say that typing a Tab key should advance the cursor to the next editable field. In PowerPlant, you do this by creating an LTabGroup object. If you examine the hierarchy window in Constructor for this PPob, you can see the LTabGroup positioned in the hierarchy. To make a tab group object, choose the **Make Tab Group** item from the Constructor's **Arrange** menu.

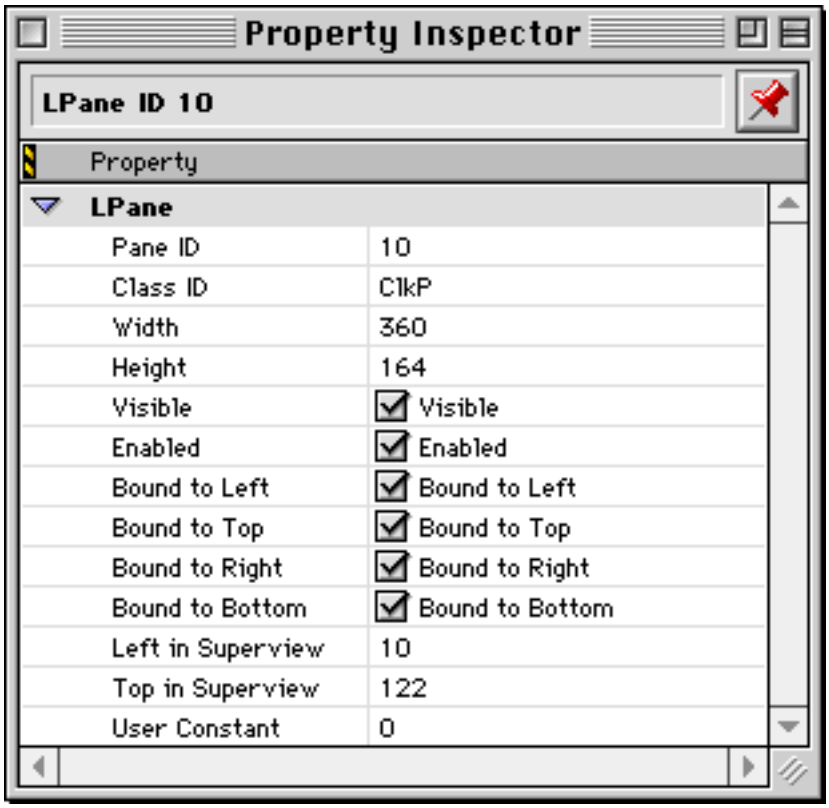
All of these are standard PowerPlant panes. The interface also includes a custom pane. This one, you build yourself.

1. **Create a custom pane.**

   If you have not already opened the Constructor project, double-click the Panes.ppob file in the IDE project window. Constructor launches and the Constructor project window appears. Double-click the LWindow view to see its contents. Also, make sure the Catalog window is open.

   Drag an LPane object from the Catalog window and drop it on the Panes window. Double-click the object to open the property inspector window so you can set its characteristics as shown in Figure 6.9.

Figure 6.9 Setting LPane properties



The pane is bound to all four sides of the window. It is enabled and visible. The Pane ID is 10.

2. **Set the class ID.**

You are creating a new class of object, and it must have its own class ID. Change the class ID from the default value “pane” to a new value, “ClkP”. The value “ClkP” is arbitrary. It is an acronym for click pane. You’re going to write code so that this pane responds to a click in a special way.

Save your changes, close all the Constructor windows, and return to the CodeWarrior IDE. It’s time to write some code.

Implementing a Custom Pane

All the panes except for the custom pane are standard PowerPlant objects. You don't have to write *any* code to make them work properly. The application-level code that creates the window has been provided for you. You'll work with applications in Chapter 9, and windows in Chapter 11.

In the remaining steps, you write the code to implement the custom pane. This is going to be a lot easier than you might think.

3. **Declare the class ID.**

`Class declaration` `CClickPane.h`

Because this is the first step, we'll point out the code locator. Open the `CClickPane.h` file, and locate the class declaration. Most of the header has been provided for you. We assume you know what a class is, and the C++ syntax necessary to declare one.

PowerPlant relies on each pane class having a unique class ID. In Step 2 you specified "ClkP" as the class ID for the CClickPane class. Each pane class has a `class_ID` enumerated constant. Declare that constant to have the value ClkP.

```
class CClickPane : public LPane {
public:
  enum { class_ID = 'ClkP' };
```

4. **Study the class declaration.**

`Class declaration` `CClickPane.h`

Look at the rest of the code in this class declaration. The CClickPane class inherits from LPane. There are several constructors and a destructor. The class also declares a new function, `DrawPaneStats()`.

Finally, the class overrides three inherited functions: `DrawSelf()`, `ClickSelf()`, and `AdjustCursorSelf()`. You write each of these functions in subsequent steps.

Save your changes and close the header file.

5. **Write the stream constructor.**

   `CClickPane(LStream *)` `CClickPane.cp`

   Building a CClickPane object does not require any data other than the data used to describe a simple LPane object. Therefore, CClickPane can use the LPane stream constructor as its own stream constructor.

   Call the LPane stream constructor.

```
CClickPane::CClickPane(LStream *inStream )
    : LPane( inStream )
{
}
```

   The other constructors and the destructor are all empty and are provided for you.

   You have done everything you need to build a CClickPane object. You have written both the class declaration and the stream constructor that builds the object from the stream. PowerPlant will call this function automatically as it reads the PPob resource. The CClickPane object will be built based on the information you set in Constructor.

   In the remaining steps you implement CClickPane functionality.

6. **Draw the pane.**

   `DrawSelf()` `CClickPane.cp`

   This function should draw a frame around the pane, and then draw the contents. You are free to draw anything in the pane that suits your fancy.

   In the solution code, this pane displays statistics about itself. You can use the `DrawPaneStats()` function for this purpose. This function is provided for you, because it does not directly involve PowerPlant.

```
CClickPane::DrawSelf()
{
  // Calculate the frame rect.
  Rect theFrame;
```

```
  CalcLocalFrameRect( theFrame );

  // Draw a frame around the pane.
  ::FrameRect( &theFrame );

  // Draw the pane stats.
  DrawPaneStats();
}
```

Feel free to examine the `DrawPaneStats()` function if you wish. It reads various data members from the object, converts the values to string representations, and draws the strings.

TIP: The StTextState class used in `DrawPaneStats()` preserves and restores existing text settings. You can read about this class in "UDrawingState" on page 560.

7. **Respond to a click.**

   `ClickSelf()` `CClickPane.cp`

   This function is called whenever the user clicks an enabled pane. Write code to make something happen when the user clicks this pane. The solution code beeps. Again, feel free to experiment.

```
CClickPane::ClickSelf(const SMouseDownEvent &inMouseDown )
{
#pragma unused( inMouseDown )
  ::SysBeep( 9 );
}
```

The input parameter is unused in the solution code. If you have compiler warnings on, you may get a warning. You can avoid the warning using the `#pragma` as shown.

8. **Adjust the cursor.**

   `AdjustCursorSelf()` `CClickPane.cp`

   When the user moves the mouse into your pane, you may want to adjust the cursor. In PowerPlant, you don't have to worry about

when to do this. The framework calls `AdjustCursorSelf()` at the right time.

In this function you should get a cursor and display it. The solution code uses a cursor provided in the project resources. The constant `rCURS_Finger` is defined at the beginning of this source file. It represents the resource ID for a finger cursor.

```
CClickPane::AdjustCursorSelf( Point inPortPt,
                              const EventRecord &inMacEvent )
{
#pragma unused( inPortPt, inMacEvent )

  // Get the cursor.
  CursHandle theCursH = ::GetCursor( rCURS_Finger);

  // Set the cursor.
  if ( theCursH != nil )
    ::SetCursor( *theCursH );
}
```

Save your work and close the file.

9. **Add header file**

`Top of File` `CPaneApp.cp`

To register any PowerPlant or custom class, you need to include the header file for that class in your main source file.

```
// Custom class headers
#include "CClickPane.h"
```

10. **Register the new class.**

`CPaneApp()` `CPaneApp.cp`

The Step 5 instructions said that PowerPlant calls the stream constructor automatically. This only happens if you register your class with PowerPlant. We don't actually discuss registering classes until

"Register PowerPlant Classes" on page 288. However, it must be done for your custom class to work.

Put the new code in this step after the existing call to `RegisterClass_(LTabGroup)` in the CPaneApp constructor.

```
RegisterClass_(LTabGroup);
// Register custom classes.
RegisterClass_(CClickPane);
```

11. Build and run the application.

Make the project and run it. If it doesn't build, examine the steps and your code carefully to see where things went wrong. If all else fails you can use the solution code.

When the project builds correctly and you run the application, a window appears containing all the panes. See Figure 6.8 on page 160. Play with the window and watch what happens.

Resize the window. Watch what happens to the various panes as you do. The pane binding determines the result.

Click on the various captions, and nothing happens. These panes are not enabled. However, the editable text fields are fully functional. Try them out! You can enter text, cut, copy, paste, and so forth.

Press the Tab key to cycle through the three editable text fields. Press Shift Tab to cycle backwards. This is the automatic behavior of the invisible LTabGroup object in action. The fact that this works has nothing to do with the group box. The group box is just an aesthetic feature. It does not group the items functionally.

Observe the custom pane field. If you implemented the solution code, the pane statistics appear in the field. Move the mouse until the cursor is over the pane. The cursor should change to the finger cursor. Click the pane, and it should beep. This is your code at work: `DrawSelf()`, `ClickSelf()`, and `AdjustCursorSelf()`.

PowerPlant takes care of calling your functions at the appropriate moment. You take care of implementing the functionality.

When you are through observing, quit the application.

Congratulations! You have implemented several different kinds of standard PowerPlant panes, as well as a completely new custom pane. You're on your way! In the next chapter we move on to panes that can contain other panes—the PowerPlant view classes.

7

Views

In this chapter we're going to talk about views—LView objects and objects that descend from LView—in detail. Like the preceding chapter on panes, we'll discuss background information on views first, and then go into detail about how to use views in your application.

The principal topics in this chapter are:

- What Is a View—including the different kinds of views in PowerPlant.
- View Characteristics—a detailed look at the things that make a view a view.
- Working With Views—how to make and use views.
- Some Specific Views—details on some view classes.

After we complete this discussion, you'll create and manipulate real views in this chapter's coding exercise.

What Is a View

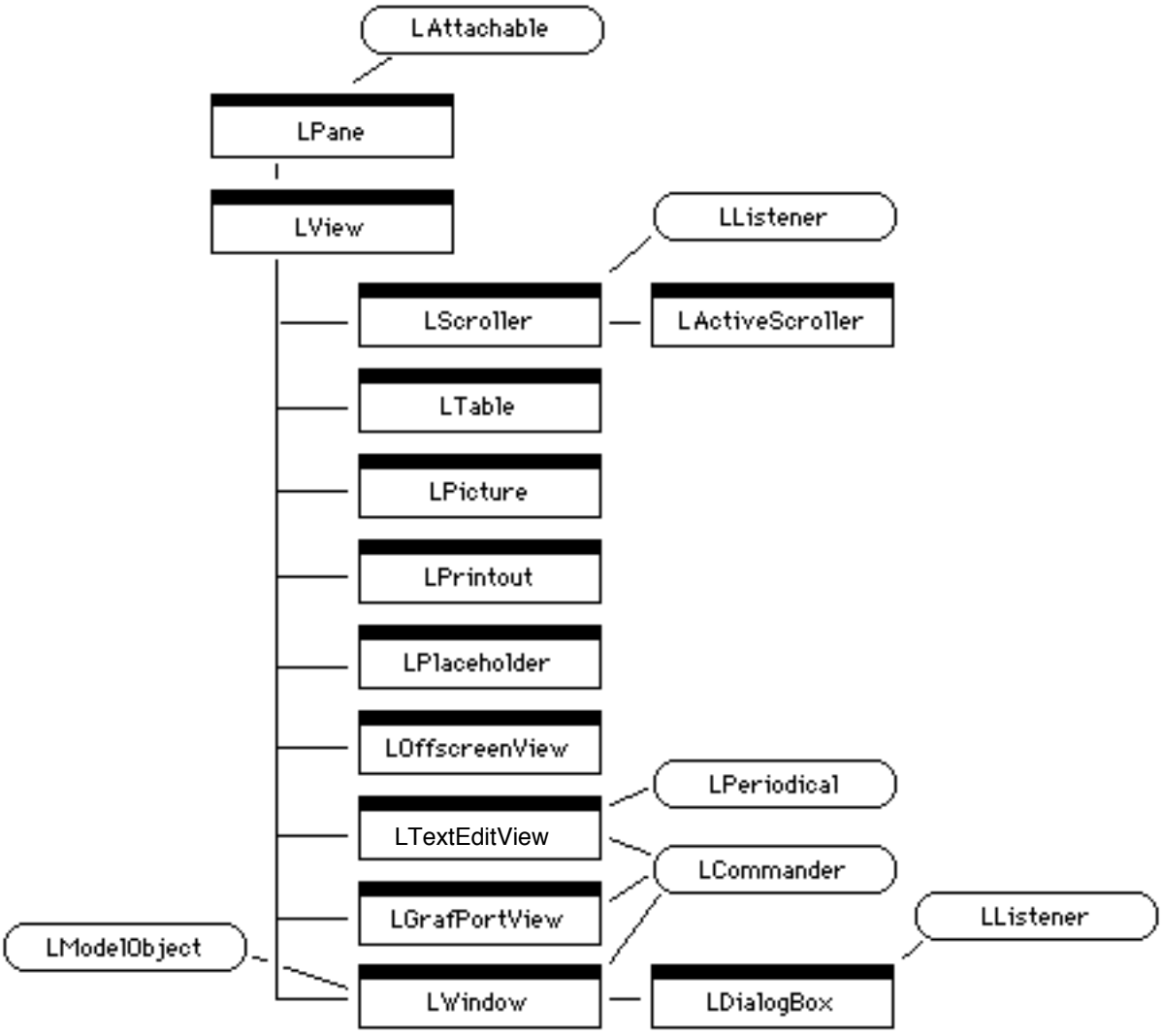
Earlier in this manual we discussed views in a general sense as a concept in an application framework. From now on, we use the term "view" to mean an object that descends directly or indirectly from LView.

As you know, a view is a special kind of pane—one that can contain another pane. As a result, the LView class is the fundamental class for managing the visual hierarchy in an application.

Figure 7.1 illustrates the inheritance hierarchy for LView and its descendants. (Remember, this is the class hierarchy, not the visual hierarchy you create in an application. To keep these ideas distinct,

we will always refer to the "visual hierarchy" when talking about how panes relate to each other on screen.)

Figure 7.1 View class hierarchy



In a PowerPlant application, you will use several kinds of views, including LView itself. You use a plain vanilla LView object as a way of grouping panes.

Of the subclasses, LWindow is the most significant and complex. We will discuss LWindow in great detail in Chapter 11, "Windows."

NOTE: Programmers familiar with other frameworks (such as MacApp) may wonder why there is a distinction between panes and views in PowerPlant. Panes encapsulate the behavior of "something that appears in a window." Views extend that behavior to "…and includes other panes." By factoring these behaviors into different classes, LPane is kept as lean as possible. Behavior for managing lists of subpanes is restricted to LView and its descendants.

The LView class encapsulates a generic interface for all view objects. As a result, all views share certain common characteristics.

View Characteristics

Remember that all views are also panes. Therefore, everything we said in Chapter 6, "Panes" applies to views as well. They have an ID number, a frame, frame binding, contents, state, and mouse information. All of the data members and member functions discussed in that chapter apply equally to views.

In this section we discuss the additional features of views that make them different from panes. They are:

- Subpanes—maintaining the visual hierarchy
- Image—the content area of the view
- Scrolling—moving the image within the frame
- Coordinate Systems—keeping track of the numbers

To see how some of these characteristics are reflected in Constructor, refer to Figure 7.7 on page 181.

Subpanes

A view may contain an arbitrary number of other views and panes. To support this characteristic, each view has an `mSubPane` data member. This is an LArray object that maintains the list of subpanes.

PowerPlant uses this data member to step through all subpanes for various purposes such as modifying state, getting an ID number, and so forth. For example, when you call `FindPaneByID()`, PowerPlant walks through the subpane list in search of the pane.

You can modify the subpane list dynamically at runtime. We'll talk about the details in "Managing Subpanes and the Visual Hierarchy" on page 187.

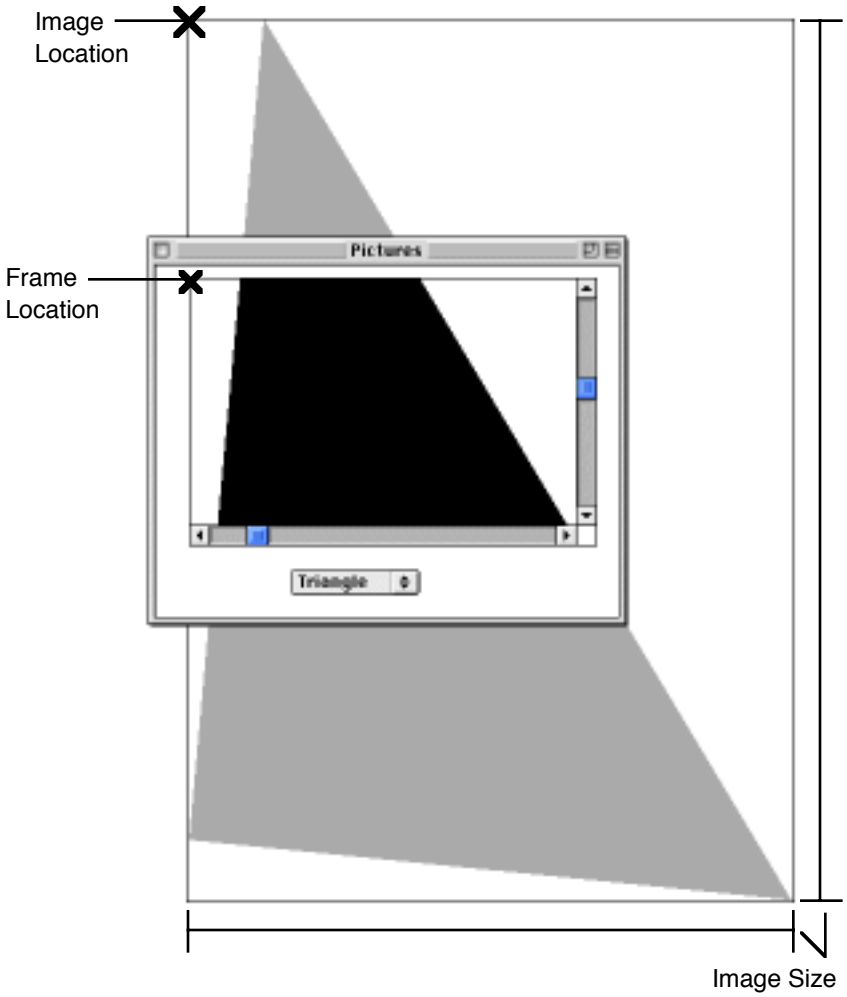
Image

In addition to a frame, a view also has an *image*. Like the frame, the image is specified by a location and a size. The image defines a rectangular area that holds the contents of the view—a picture for example, other views, other panes, or a combination of these and other objects.

The image may be larger than the frame, in which case only a portion of the image is visible in the frame. A view allows you to scroll the image area around in the frame. We will revisit scroll bars in detail in the section on scrolling, below.

Figure 7.2 shows the relationship between a frame and an image.

Figure 7.2 Image and frame in a view



In Figure 7.2, we're looking at the image area of a view that is attached to a scroller, which in turn is contained in a window view.

Scrolling

Because the frame and image may not be the same size, views support scrolling so that you can see different parts of the image. In practice, the only views you'll use for scrolling are LScroller and its subclass, LActiveScroller. From now on we will call these "scroller views," or simply "scrollers."

The most important feature of a scroller view is that it has one and only one subpane. That subpane is a view. The subview may contain an arbitrary number of panes and views. The net effect is that the scroller view may contain (through the intermediary subview) any number of panes of any type. We will call the single subview contained inside the scroller the "scrolling view" because it is the view that moves around. Remember, there is a distinction between the scroller view (that manages the scroll bars and coordinate system adjustments) and the scrolling view (that simply appears inside the scroller).

WARNING! Make sure your scroller has only one subpane! To put multiple items in a single scrolling view, use LView as the subpane to the scroller, and put the multiple items in the LView.

Scroller views implement all the functionality of scroll bars with which you have become familiar. They create, maintain, resize, move, hide, show, enable, disable, and manage everything having to do with scroll bars. The only difference between LScroller and LActiveScroller is that the latter supports dynamic scrolling of the view. Dynamic scrolling means that the window contents scroll while the user drags the scroll bar thumb.

Coordinate Systems

Views are responsible for transforming coordinates among four different coordinate systems:

- Global coordinates
- Port Coordinates
- Local Coordinates
- Image Coordinates

When you derive classes and create code to draw your own panes, you will likely face situations where you must convert coordinates. You should understand what the coordinate systems are so that when you do need to manage coordinates, you'll know what's

going on. PowerPlant pane and view classes have a series of coordinate conversion routines you can use to do the work.

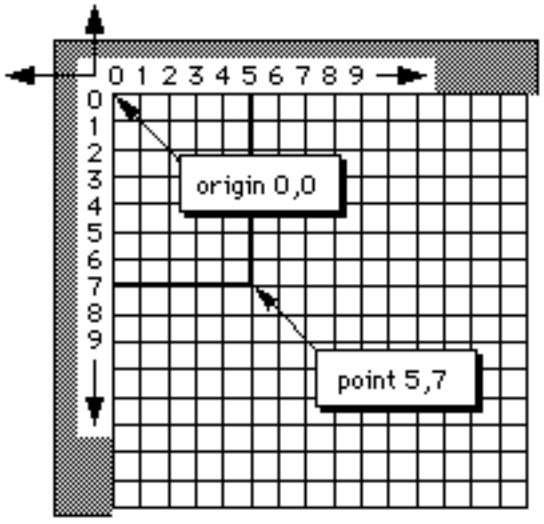
We'll discuss the individual routines in "Working With Views" on page 180.

NOTE: Most of the coordinate transformation routines are available to all panes, not just views. However, in the non-view classes the functions simply call the superview's coordinate transformation routines. Therefore, all real work happens in the view objects.

Global coordinates

The **global coordinate system** has its origin (0,0) at the top left corner of the main monitor. This is the monitor with the menu bar. The range of coordinate values is -32,768 to 32,767 (a signed 16-bit integer).

Figure 7.3 Global coordinates



We will call this area "QuickDraw space." At the standard 72 dots-per-inch screen resolution, QuickDraw space is a square about 76' (23 m) on each side.

QuickDraw uses global coordinates to determine the positions of multiple monitors in QuickDraw space, and to position windows. Monitors or windows to the left or above the top left corner of the main monitor would have negative coordinates.

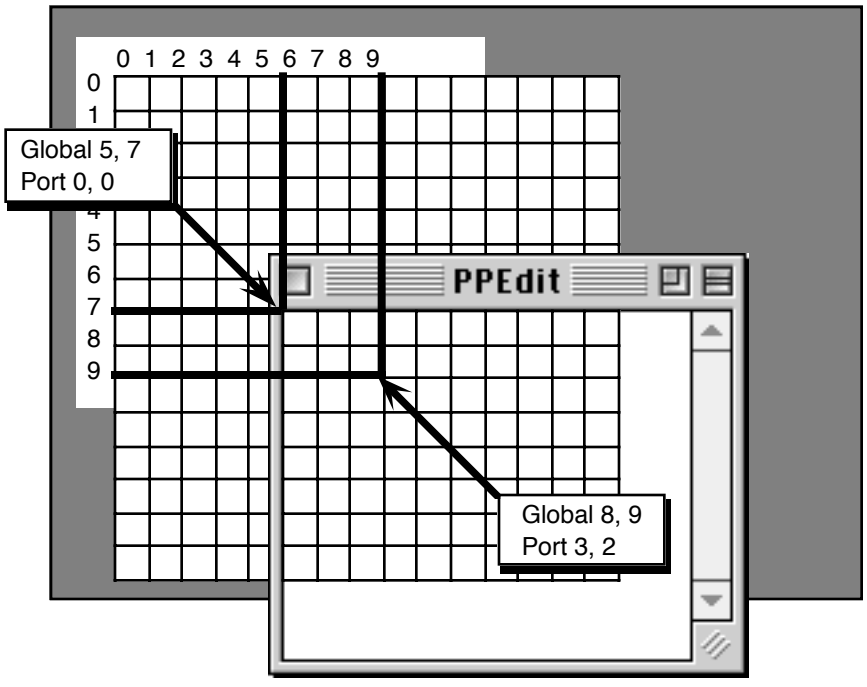
You use this coordinate system when you use a Toolbox routine that requires global coordinates.

Port Coordinates

In the **port coordinate system**, the origin (0,0) is the top left corner of the content space of a GrafPort (usually a window). This system is also called the *window coordinate system*.

Except for the fact that it has a different origin, the port coordinate system is identical to the global coordinate system, as shown in Figure 7.4.

Figure 7.4 Port coordinates



Note that there is a separate port coordinate system for each open GrafPort. Each port has its own top left corner, and that's the (0,0) point for its internal coordinate system. That same point also has a global value, which can be any location in QuickDraw space.

In traditional Macintosh programming, most of your work is done in port coordinates. In PowerPlant, you use this coordinate system to specify the location of panes within a window. Therefore, the location data for the frame of each pane is in port coordinates, and represents the distance from the top left corner of the port.

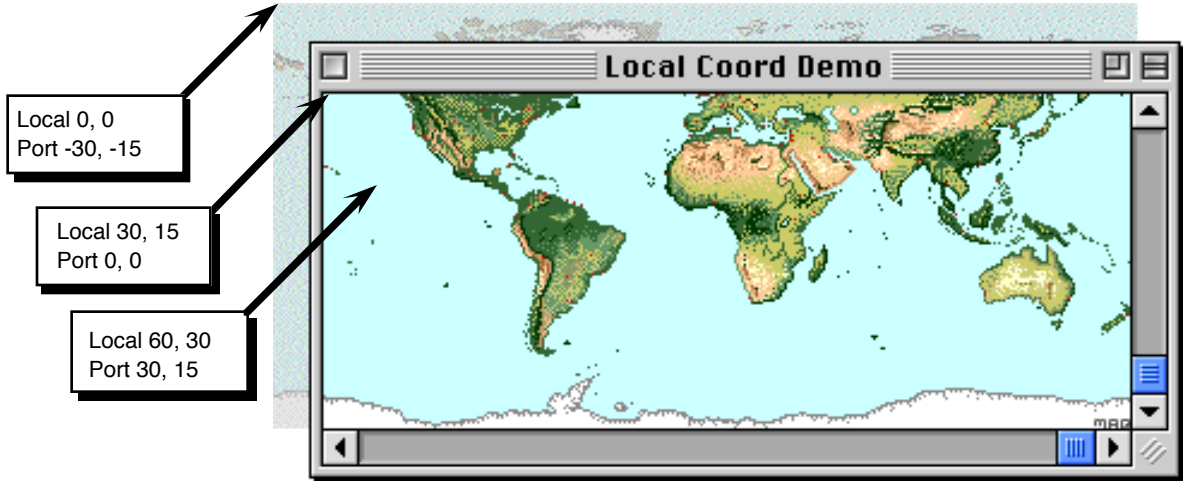
PowerPlant uses this coordinate system internally to maintain the spatial relationships among panes and views.

Local Coordinates

In PowerPlant, the origin of *local coordinates* is the top left corner of the image in a view. Local coordinates are 16-bit values in QuickDraw space. Figure 7.5 displays this relationship. There is a scrolling view in part of the window. Parts of the view's image extend beyond the view frame and would normally be invisible. We have rendered it gray for illustrative purposes. The view frame has a heavy border

In Figure 7.5, the top left corner of the view frame is at location (50,15) in port coordinates. The image in a view may be scrolled, so the top left corner of the image is not necessarily the same as the top left corner of the view frame. Local coordinates measure from the top left corner of the image, not the view frame. In Figure 7.5, the location of the frame is (30,10) in local coordinates.

Figure 7.5 Local Coordinates

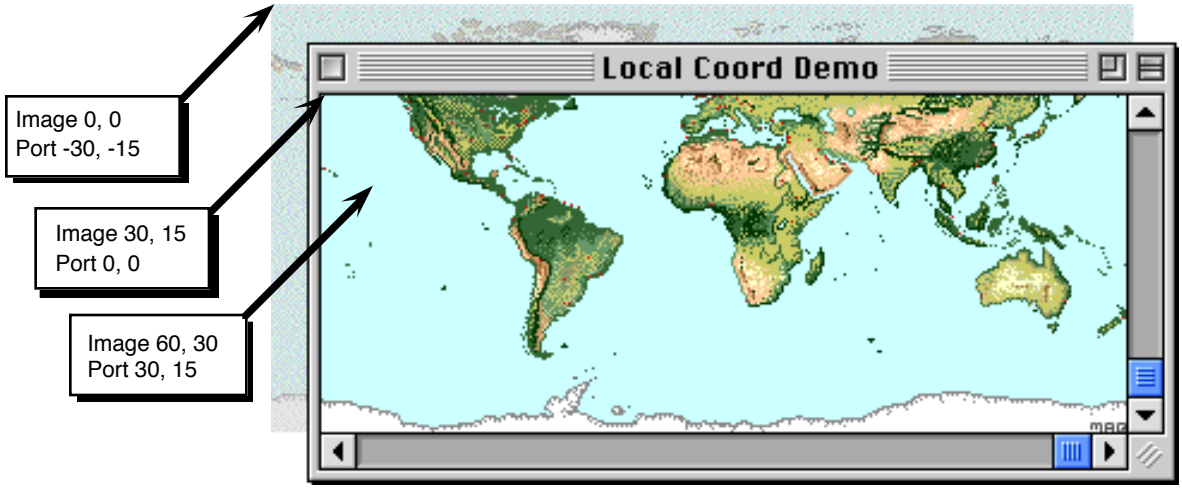


Each view has its own local coordinate system. This means that you can draw in a view without worrying about the view's location in a window or in another view. PowerPlant takes care of most of the necessary coordinate transformations automatically.

Image Coordinates

The image coordinate system is PowerPlant-specific. The **image coordinate system** uses 32-bit values in which the top left corner of the image is (0,0), just like local coordinates.

Figure 7.6 Image coordinates



You may have noticed that the illustration for image coordinates is virtually identical to that for local coordinates. At values less than 16K, local coordinates and image coordinates are identical. Most applications do not use a drawing space larger than 16K pixels.

If you do need a large drawing space, the image coordinate system provides the support you need to scroll an image larger than that allowed in standard QuickDraw space.

PowerPlant uses positive numbers for image coordinates. The drawing area ranges from 0 to 2,147,483,647 to the right and down from the origin. At a resolution of 72 dpi, this is a square more than 470 miles (757km) on a side.

WARNING! If a view's image size is larger than 16K pixels, you must convert image coordinates to local coordinates before drawing. We'll discuss this problem in detail in "Managing Coordinate Transformations" on page 192.

Working With Views

In this section we discuss how to use PowerPlant for view-related tasks. We'll talk about:

- Creating a View—Constructor, and using view constructor functions.
- Drawing a View—drawing and updating views.
- Managing Subpanes and the Visual Hierarchy—adding and removing subpanes.
- Managing the View Image—adjusting location and size.
- Managing Scrolling—how to scroll a view.
- Managing Coordinate Transformations—when and how to convert coordinates.

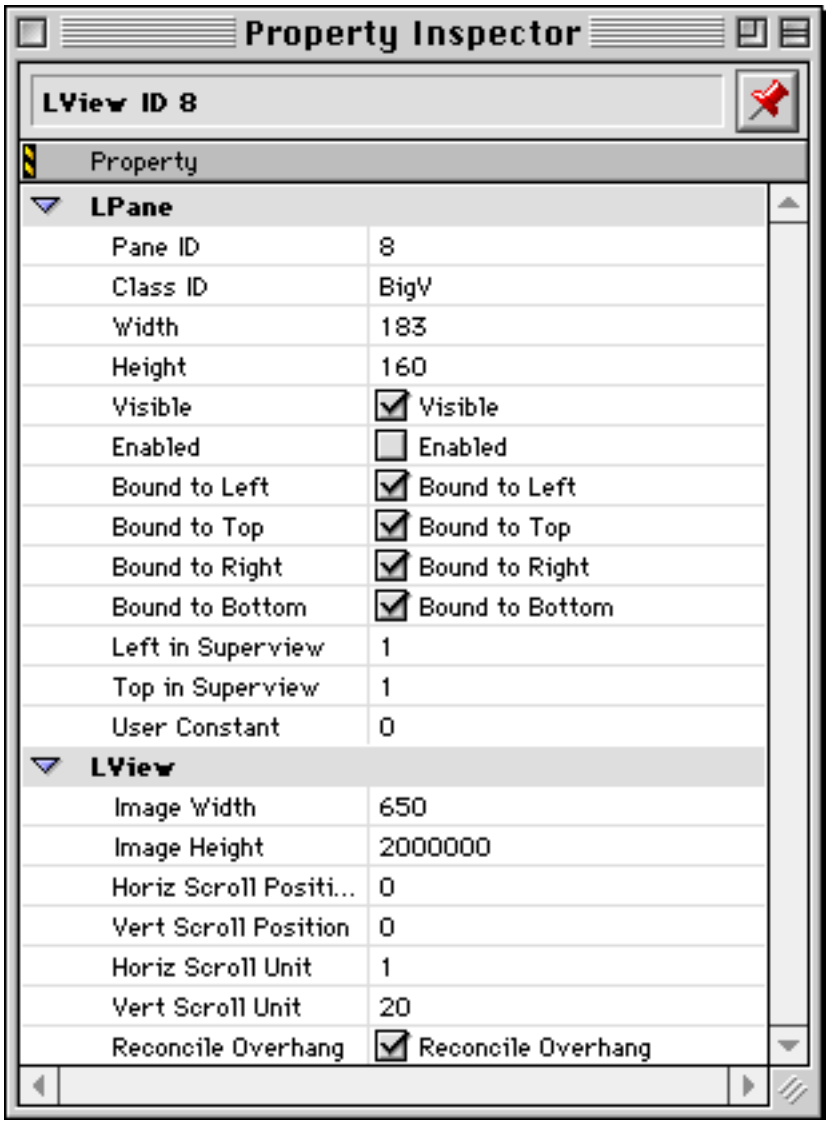
Creating a View

You can create a view using Constructor, or on the fly in your code. We talk about each method. Then we discuss what you do when you derive your own class from LView or its descendants.

Using Constructor

Creating a view object in Constructor is simple. While in Constructor, you drag a view object from the tool palette into a containing view. Then you set the characteristics for that object. Figure 7.7 shows the Property Inspector window for a simple view.

Figure 7.7 Creating a view in Constructor



Note that a view has all the same information as a pane, including location, size, pane ID, class ID, binding, and state information. Remember, when you derive your own classes you must change the class ID to your own unique value and register the class with PowerPlant before creating any objects of that class.

Each view has its own location in its superview, specified by the top left coordinates. The coordinates for any pane you place inside the

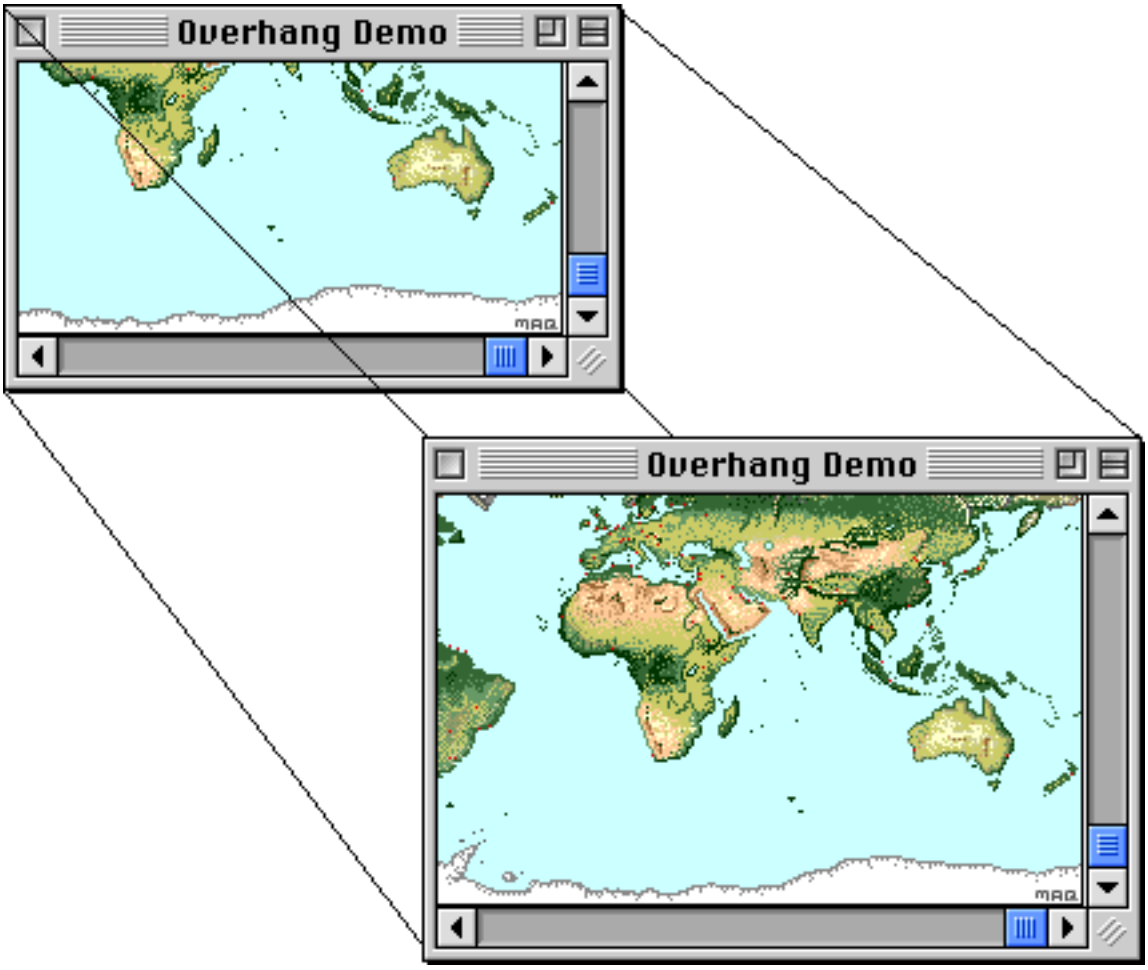
view are relative to the view's top left corner. If you change the pane's superview, you may need to change the pane's location as well. If you don't, you may encounter some seemingly odd behavior. Here's an example.

Assume you design a window, and place a pane at coordinate 200, 300. You subsequently decide to create a new view inside the window to contain the pane, and locate that view at 190, 290. If you simply move the original pane to its new view, it appears at coordinate 200,300 *in the view*, or 390, 590 in the window. If you want the pane to appear at position 10,10 in the new view (200,300 in the window), you must change the pane's location to 10,10.

The scroll unit is the number of pixels to scroll when the user clicks in the scroll arrow. The scroll position is the starting point of the scroll when the view is created, typically zero.

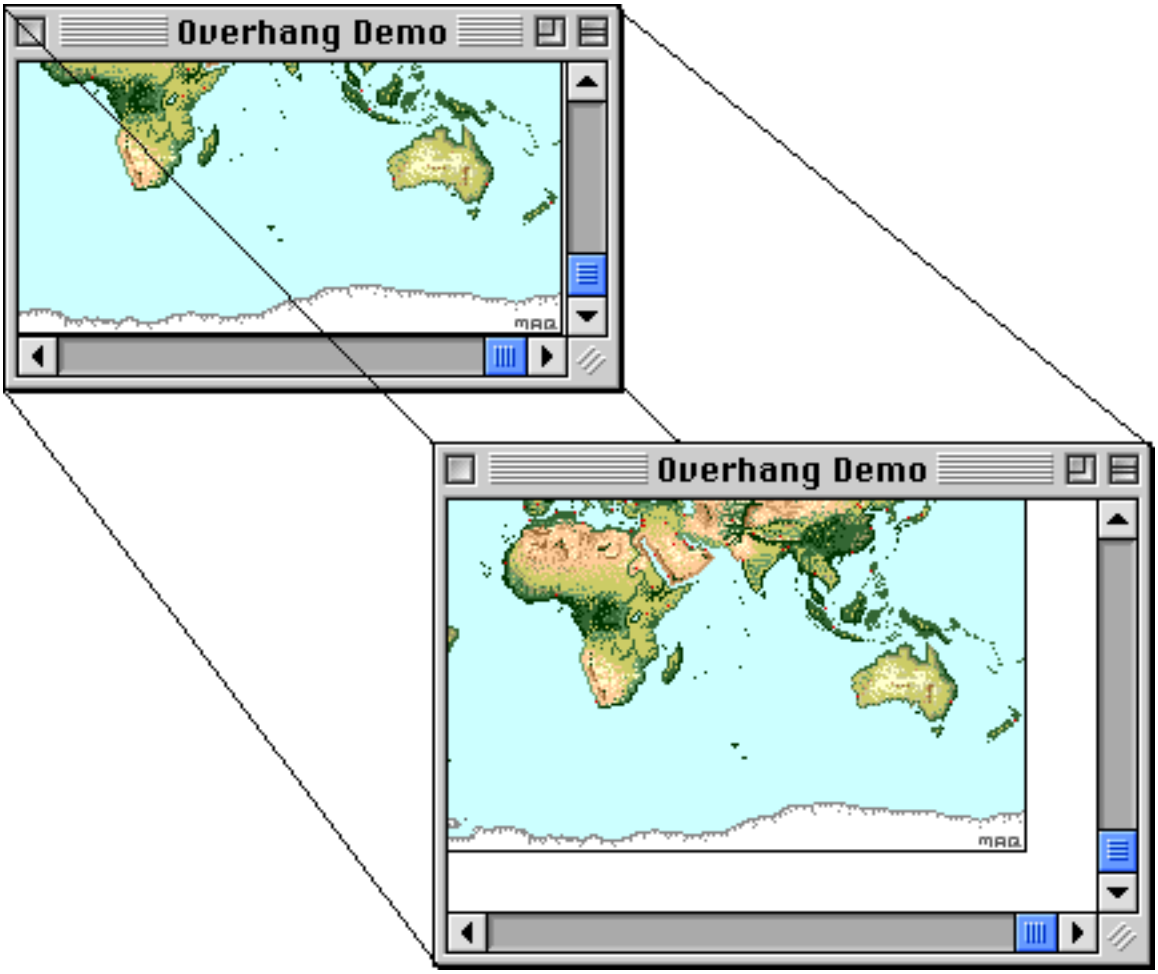
The Reconcile Overhang check box (see Figure 7.7 above) controls how the view's contents scroll when the superview is resized beyond the bottom right corner of the contents. If Reconcile Overhang is on (Figure 7.8), the contents reposition so that the bottom right corner of the contents remain at the bottom right corner of the window.

Figure 7.8 Reconcile overhang on



If Reconcile Overhang is off, then the contents do not reposition, as shown in Figure 7.9.

Figure 7.9 Reconcile overhang off



Other view classes have additional characteristics. We'll examine some of them in "Some Specific Views" on page 194. Remember, the general process for creating any view object with Constructor is: drag the item into the view, set its position in the view hierarchy, then set its characteristics.

NOTE: When you create a new PPob resource in Constructor, you automatically get a new view. You choose from LWindow, LDialogBox, LPrintout, LGrafPortView, or LView. This is the top

view of the visual hierarchy, and it has no superview. You must set the characteristics for this view as well.

See also "Register PowerPlant Classes" on page 288.

Creating a view on the fly

The typical approach used when creating a view object on the fly is to define both an `SPaneInfo` structure and an `SViewInfo` structure. The `SViewInfo` structure in Listing 7.1 specifies the values required to build a generic view. You then call the appropriate constructor. Depending upon the particular view you are creating, you may need to pass additional parameters.

NOTE: The LWindow and LDialogBox classes use a different structure, `SWindowInfo`. See "Creating a window on the fly" on page 366 for more information.

Listing 7.1 The SViewInfo structure

```
typedef struct SViewInfo {
  SDimension32  imageSize;
  SPoint32      scrollPos;
  SPoint32      scrollUnit;
  Int16         reconcileOverhang;
} SViewInfo;
```

Note how the fields in this struct parallel the characteristics you provide in Constructor, as shown in Figure 7.7 on page 181.

Each view class has specific constructors of course, one of which (except for windows) receives pointers to the `SPaneInfo` and `SViewInfo` structures. Most have additional parameters you must provide. Use the *PowerPlant Revealed* HTML reference to learn details on the various constructors and the parameters you must provide to successfully create an object on the fly.

TIP: For code demonstrating window and pane creation on the fly, see the Panes Demo code in the `CodeWarrior Examples:Mac OS Examples:PowerPlant Examples:Updated Examples` folder.

After you have created a view and installed all its panes, you should call `FinishCreate()`. This function ensures that each pane's state (visible/invisible, active/inactive, enabled/disabled) matches its superview.

You are also responsible for maintaining the visual hierarchy. If the new view will have a superview, call `SetDefaultView()` for the container view *before* creating the new view with a stream constructor. This ensures that when you create the new view it has the correct superview. Alternatively, you can call `PutInside()` as described in "Managing Subpanes and the Visual Hierarchy" below.

Similarly, if the new view is also a commander (LWindow, LDialogBox, LTextEditView, LGrafPortView, or your own derived class), you must maintain the command hierarchy as well. Call `SetDefaultCommander()` before creating the view.

See also "Creating a pane on the fly" on page 138 for more on the SPaneInfo structure, "Managing Subpanes and the Visual Hierarchy" on page 187, and "Command Chain" on page 309 for more on the command hierarchy.

Deriving your own views

When you derive a class from LView or one of its descendants, you typically define a destructor and several constructors: a default constructor, a constructor to build the view from `SPaneInfo` and `SViewInfo` structures, a constructor to build the view from a stream, and a copy constructor. For more on stream constructors, see "Stream constructor" on page 141.

NOTE: The LWindow and LDialogBox classes use a different structure, SWindowInfo. See "Creating a window on the fly" on page 366 for more information.

When you derive your own class, you may of course override whatever functions are necessary for your needs. We'll discuss the specific details of deriving window classes in "Deriving your own windows" on page 367.

See also "Creating a view on the fly" on page 185 for more on the SViewInfo structure.

Drawing a View

The typical view is a container for panes, and the panes draw themselves. Therefore, drawing a view requires nothing more than calling the view's `Draw()` function.

In most cases, the default behavior provided in PowerPlant will suffice for your needs. `Draw()` performs housekeeping details, calls the view's `DrawSelf()` function, walks through the list of subpanes, and sends each subpane a `Draw()` message.

Views, being panes, have a `DrawSelf()` function that you can override if necessary to do view-specific drawing. For example, an LWindow object may erase itself before drawing all its subpanes. Note that the `DrawSelf()` function is called before subpanes draw, because the view is visually behind its contained subpanes.

TIP: Remember that if you do any drawing in a view that does not go through the `Draw()` function, you must call `FocusDraw()` yourself to ensure that the port is set up correctly.

Managing Subpanes and the Visual Hierarchy

All pane classes have member functions for adding themselves to a superview (going up the tree). Use `PutInside()` to make any pane or view a subpane to a superview. To remove a pane or view from a superview, call `PutInside()` and pass nil as the new superview. `PutInside()` is the only function you need for complete visual hierarchy maintenance.

You do not need to work from the view level going down. If you have a group of panes and you want to add them to a view, you do not tell the view to add the panes. You tell each pane to add itself to the view. The view is a container, the panes do the work.

There is no member function to identify the topmost container in a visual hierarchy, typically a window. The code in Listing 7.2 shows you one way to do this. It assumes you have a pointer to an LPane object in a variable, `inPane`. When this code completes, the local variable `theTopView` holds the topmost view in the visual hierarchy.

Listing 7.2 Finding the topmost view

```
LView* theTopView = inPane->GetSuperView();
if (theTopView != nil)
{    // follow chain of superviews to top
    while (theTopView->GetSuperView() != nil)
    theTopView = theTopView->GetSuperView();
}
else // thePane is the top
{
  theTopView = dynamic_cast<LView*> (inPane);
}
```

View classes have several functions for dealing with their subpanes, as listed in Table 7.1.

Table 7.1 Subpane management functions

Function	Purpose
`GetSubPanes()`	returns a reference to the LList object containing all subpanes for the view
`OrientSubPane()`	sets subpane state to match view
`ExpandSubPane()`	resize the subpane to fill the view horizontally and/or vertically
`DeleteAllSubPanes()`	destroy all subpanes

Function	Purpose
`FindPaneByID()`	given an ID, return a pointer to the pane
`FindSubPaneHitBy()`	find subpane of this pane that contains the point
`FindDeepSubPaneContaining()`	search through subpanes for deepest pane containing the point
`FindShallowSubPaneContaining()`	search through subpanes for shallowest pane containing the point
`GetValueForPaneID()`	given a pane ID, return the pane's value
`SetValueForPaneID()`	given a pane ID, set the pane's value
`GetDescriptorForPaneID()`	given a pane ID, return the pane's descriptor
`SetDescriptorForPaneID()`	given a pane ID, set the pane's descriptor

The last four functions in Table 7.1 are accessors for the same value and descriptor information we discussed in "Value and descriptor" on page 133. These functions simply find the pane by ID number and call the appropriate accessor.

Managing the View Image

Like the frame in a pane, you can adjust the image in a view. Table 7.2 lists the available functions and their purpose.

Table 7.2 Image and frame management function

Function	Purpose
`GetImageSize()`	return image size
`GetImageLocation()`	return image location

Function	Purpose
`ResizeImageTo()`	set new size to absolute value
`ResizeImageBy()`	set new size to relative value
`CalcRevealedRect()`	calculate the portion of the frame that is visible through the frames of all superviews (in port coordinates)
`GetRevealedRect()`	return the revealed rectangle

Note that if the image is connected to a scroller, resetting the image size means you should update the minimum and/or maximum values of the scroll bars by calling the scroller's `AdjustScrollBars()` member function.

Managing Scrolling

All view classes have certain functions required for scroll management. Under normal circumstances, you should never have to deal with these functions with the possible exception of three accessors.

Table 7.3 Scroll-related accessors

Function	Purpose
`GetScrollPosition()`	return location of a view's frame within its image
`GetScrollUnit()`	return the current value of the scroll unit
`SetScrollUnit()`	set the value of the scroll unit

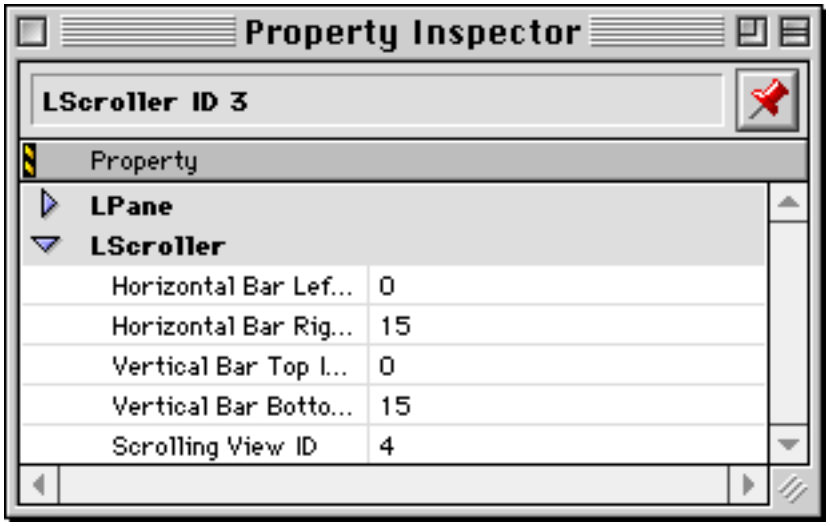
For all practical purposes, scrolling is limited to two classes: LScroller and LActiveScroller. These classes have the real functional tools you use when scrolling. There are four data members of general interest, listed in Table 7.4.

Table 7.4 Some LScroller data members

Date Type	Name	Purpose
`LView*`	`mScrollingView`	pointer to scrolling view contained in scroller view
`PaneIDT`	`mScrollingViewID`	ID of scrolling view contained in scroller view
`LStdControl*`	`mVerticalBar`	pointer to vertical scroll bar
`LStdControl*`	`mHorizontalBar`	pointer to horizontal scroll bar

When you create a scroller view in Constructor, you provide scroll-specific information, as shown in Figure 7.10.

Figure 7.10 Scroller-specific information in Constructor



The Scrolling View ID corresponds to the `mScrollingView` data member. The "indent" values are the distance from the edge of the view to the end of the scroll bar. An indent of 15 pixels at the bottom right of the view allows room for a grow box and keeps the two scroll bars from overlapping. If you don't want a scroll bar in one direction, set the left or top indent to -1.

Unless you are doing something unusual you shouldn't have to override any of the default behavior. Table 7.5 lists some of the common functions in the LScroller class.

Table 7.5 Some scroll-related functions

Function	Purpose
`AdjustScrollBars()`	adjust value, min, and max of scroll bars based on scroll and scrolling view
`VertScroll()`	scroll while clicking and holding inside the vertical scroll bar
`HorizScroll()`	scroll while clicking and holding inside the horizontal scroll bar
`VertSBarAction()`	Toolbox callback function for action while tracking click in vertical bar
`HorizSBarAction()`	Toolbox callback function for action while tracking click in horizontal bar

Managing Coordinate Transformations

No matter how helpful an application framework is, from time to time you may find yourself working at the pixel level. At that time, coordinate manipulation may become important to you. PowerPlant has routines for converting from one coordinate system to another, as listed in Table 7.6.

Table 7.6 Coordinate conversion functions

From	To Global	To Port	To Local	To Image
Global	`n/a`	`GlobalToPort Point()`	`n/a`	`n/a`
Port	`PortToGlobal Point()`	`n/a`	`PortToLocal Point()`	`n/a`

From	To Global	To Port	To Local	To Image
Local	`n/a`	`LocalToPort Point()`	`n/a`	`LocalToImage Point()`
Image	`n/a`	`n/a`	`ImageToLocal Point()`	`n/a`

Don't forget `CalcPortFrameRect()` and `CalcLocal-FrameRect()`. These functions return any pane's frame (views are panes after all) in the desired coordinate system.

Two additional functions you may find useful when working with image coordinates are `ImageRectIntersectsFrame()`, and `Im-agePointIsInFrame`.These functions determine whether a Rect or Point in image coordinates appears in the view's frame.

If the image size of a view is greater than 16K pixels, you must keep in mind the difference between image coordinates and local coordinates. As long as the image size is less than 16K, the image and local coordinates are identical for a given view. If your view's image is larger than 16K pixels, you must convert from image to local coordinates so that QuickDraw can handle the drawing.

For example, suppose that you want to draw a 10 by 10 square at the bottom right corner of the image. This is how you would do it if you know that the image fits entirely in QuickDraw space.

Listing 7.3 Drawing in QuickDraw space

```
void MyView::DrawSelf()
{
  Rect    square;

  square.top    = mImageSize.height - 10;
  square.left   = mImageSize.width - 10;
  square.bottom = mImageSize.height;
  square.right  = mImageSize.width;
```

```
  ::FrameRect(&square);
}
```

To handle an image larger than 16K pixels, the member function might look like this.

Listing 7.4 Drawing in 32-bit space

```
void MyView::DrawSelf()
{
  SDimension32   imageSize;
  SPoint32       imPos;
  Point          localPos;
  Rect           square = {0, 0, 10, 10};

  GetImageSize(imageSize);
  imPos.v = imageSize.height - 10;
  imPos.h = imageSize.width - 10;

  // if this is visible in frame
  if (ImageRectIntersectsFrame(imPos.h, imPos.v,
                               imPos.h + 10, imPos.v + 10)) {
    // convert to local coordinates
    ImageToLocalPoint(imPos, localPos);
    ::OffsetRect (&square, localPos.h, localPos.v);
    ::FrameRect(&square);
  }
}
```

Some Specific Views

Like the LPane class, LView has several direct and indirect descendants. In this section we list some of the features of each that make them unique. We will discuss the following classes:

- LGrafPortView—for use with non-PowerPlant code
- LOffscreenView—to draw off screen

- LTextEditView—display and manage editable text using TextEdit
- LTable—display and manage tabular data
- LPicture—display a PICT resource

There are other classes that derive from LView, discussed elsewhere in this manual. See also:

- Chapter 11, "Windows" for a discussion of LWindow.
- Chapter 12, "Dialogs" for a discussion of LDialogBox.
- Chapter 14, "Printing" for a discussion of LPrintout and LPlaceholder.
- "Managing Scrolling" on page 190 for information on LScroller and LActiveScroller.

LGrafPortView

An LGrafPortView object is a top-level PowerPlant view (it's superview should be nil) that can be hosted inside a non-PowerPlant window. This allows you to use PowerPlant panes in other application frameworks or in externals such as HyperCard XMCDs.

An LGrafPortView object acts as the interface between PowerPlant Panes and "foreign" code that knows nothing about PowerPlant. You are responsible for calling proper LGrafPortView functions at certain times.

See also *PowerPlant Revealed* for more on the LGrafPortView class.

LOffscreenView

An LOffscreenView is a view whose image draws off screen in a temporary GWorld and is then copied to the screen. Using an LOffscreenView can result in smoother screen updating.

This is a fairly simple extension of the standard LView class. When you draw, the LOffscreenView object creates a stack-based StOffscreenGWorld object. The view then draws its contents into the offscreen world. When the drawing operation ends, the StOffscreenG-

World goes out of scope. Its destructor uses `CopyBits()` to blit the image to the screen before destroying itself.

The typical use for LOffscreenView is as a superview for several subpanes, where you want all the panes to appear to draw simultaneously. When you draw the subpanes, they all draw in the offscreen view. When all panes are finished drawing, the complete image is blitted to the screen. Even though this is in fact no faster (and actually a trifle slower) than drawing the individual panes directly on screen, the net effect appears faster to the user because all the panes appear simultaneously. This technique is also useful if the panes overlap and might cause flicker while drawing.

LTextEditView

LTextEditView is a wrapper class for the TextEdit functionality of the Macintosh Toolbox. LTextEditView provides significant, text-related functionality using a multiple styles.

LTextEditView also inherits from LCommander so that it can respond to keystrokes, and LPeriodical to maintain cursor flashing.

Figure 7.11 illustrates additional information you provide in Constructor for an LTextEditView object.

Figure 7.11 LTextEditView-specific data in Constructor

You can specify a TEXT resource to use as the initial text. Like all text-related PowerPlant objects, you can set the font, style, size, color, and justification of the contents of the LTextEditView object using a text traits resource.

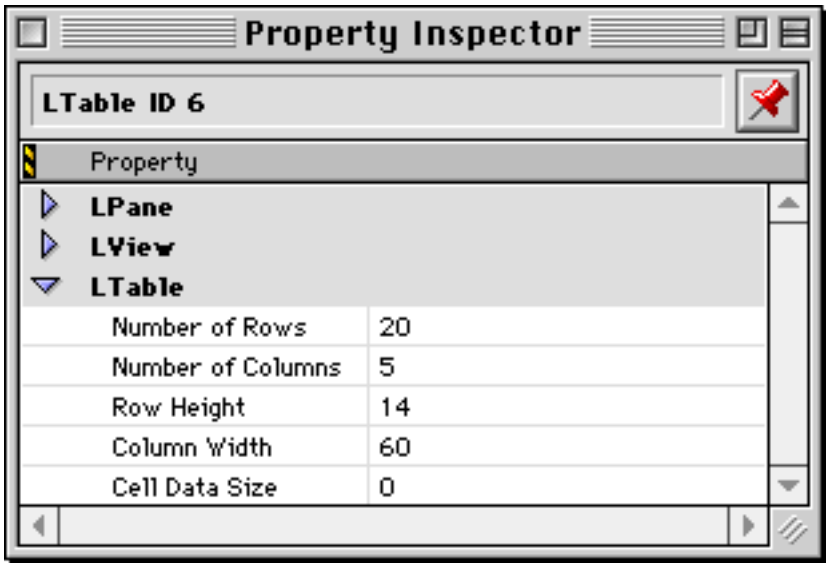
LTable

The LTable class displays and manages tabular data—rows and columns of rectangular cells in a two-dimensional grid. This is a fairly simple class that provides significant functionality.

TIP: The PowerPlant Advanced Topics manual has a chapter devoted to the display of tabular data with LTableView. If you work with tabular data, you'll find that chapter very useful.

Figure 7.12 illustrates the kind of characteristics you can specify in the LTable class. Like other classes, you can create an LTable easily in Constructor, or on the fly using constructor functions.

Figure 7.12 LTable-specific data in Constructor



Most of these characteristics are self-evident. The Cell Data Size characteristic controls how much memory is allotted for holding the tabular data, in bytes. If your data has a maximum size, you specify the size required to hold the data for a cell. PowerPlant creates an object of LArray with the number of elements equal to the number of cells in your table. You must initialize the array contents yourself.

If your data is not a known size, you can specify zero as the cell data size. You are then completely responsible for attaching data to individual cells in whatever way is appropriate for your application.

Table 7.7 lists common LTable functions.

Table 7.7 Some LTable functions

Function	Purpose
`DrawCell()`	draw the contents of a cell
`ClickCell()`	respond to a click in the cell

The default functions of the LTable class simply draw the cell's row and column number in the cell. Clicking in a cell causes it to be highlighted. However, by deriving your own class from LTable and overriding the `DrawCell()` and `ClickCell()` functions you can modify the LTable class to support the display of any kind of data and respond to clicks in whatever way is appropriate.

This makes LTable much more flexible than the Toolbox List Manager, or PowerPlant's LListBox class that uses the List Manager. At the time of this writing the LTable class does not support multiple-cell selections. You would have to add this functionality in a derived class if it is important to your application. You can also explore the LTableView class.

LPicture

The LPicture class displays a PICT resource. This is a simple extension of the LView class. The only additional feature is that the LPicture class keeps the resource ID of a PICT resource. It provides accessors to get and set the number. The LPicture class overrides `DrawSelf()` to display the picture.

This class derives from LView rather than directly from LPane because a picture's dimensions may be larger than the available frame. Therefore, the LPicture class requires the image feature of the LView class. The image bounds of an LPicture object typically match the bounds of the attached picture.

Summary

You've reached another major milestone in your understanding of the fundamental PowerPlant building blocks. You have learned all about views.

Views are the fundamental PowerPlant object for maintaining the visual hierarchy. There are many kinds of views, including LView, LWindow, LScroller, LTextEditView, LTable, LPicture, and so forth.

Views have a variety of characteristics. In addition to everything a pane has, views have a subpane list, and an image. Views also manage scrolling and coordinate systems.

You typically create a view with Constructor. You can build views on the fly in code. For views other than LWindow, you use both the `SPaneInfo` and `SViewInfo` structures as a basis, and pass any other necessary information to the appropriate constructor function. You can get, set, or otherwise manipulate every characteristic of a view at runtime should you so desire.

Like panes, this is a tremendous amount of information to absorb at one time. However, you now have a really solid understanding of LView and some of its descendants. That will make understanding windows and dialogs that much easier when we discuss them later.

Let's put all this view-related knowledge to work and see how to create and use views in code.

Code Exercise

In this exercise you build an application titled, appropriately enough, "Views." This exercise is structured very much like the exercise in Chapter 6 on panes. A PPob with most of the necessary objects has been provided for you. You will create a custom view that manages and draws in an image 650 pixels wide by 2,000,000 pixels long—about 2,300 feet (700m)!

The Interface

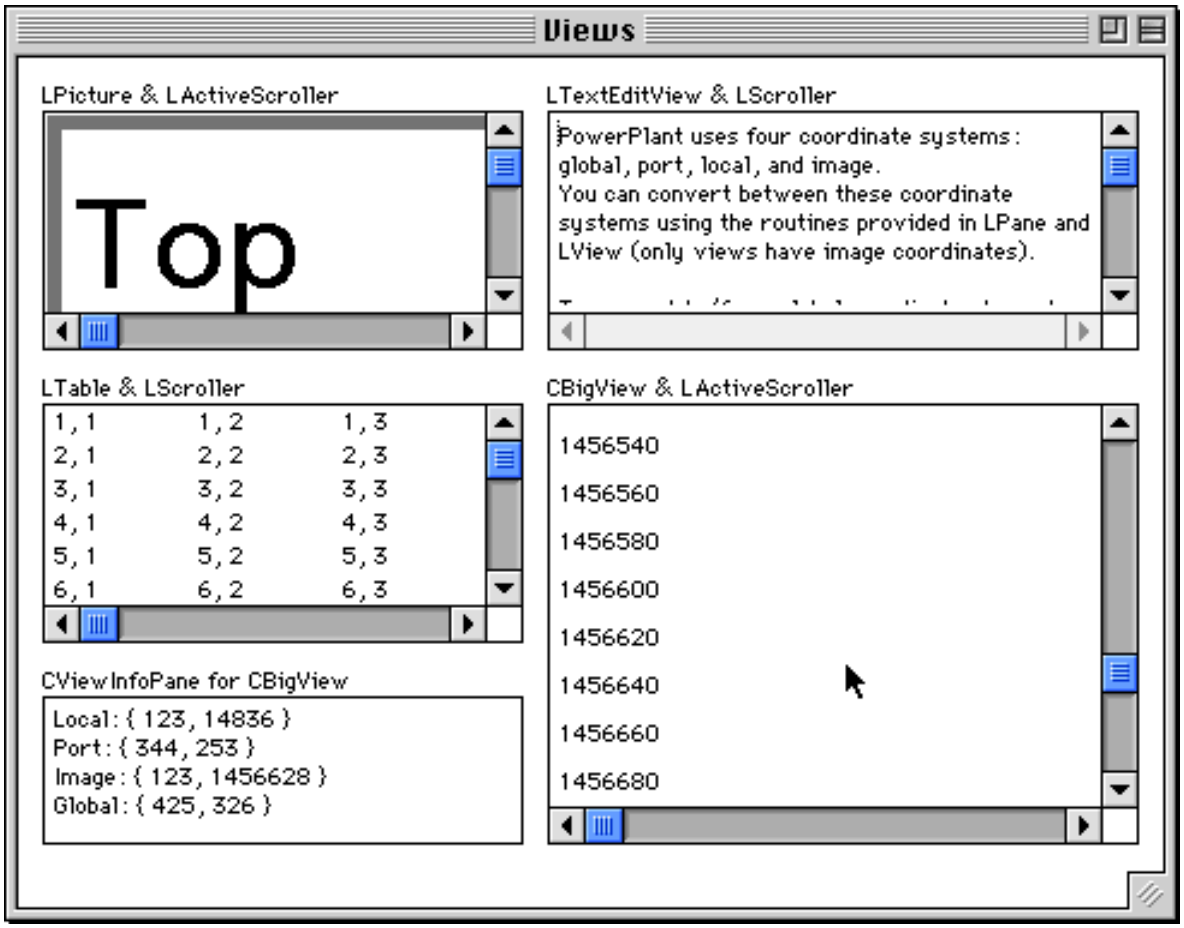
The final application looks like Figure 7.13. Take a look at the picture.

There are eight views in this window. There are four scrollers: two LActiveScroller objects, and two LScroller objects. Each scroller contains one view: an LPicture, an LTextEditView, an LTable, and a custom view, CBigView. You will write the code for CBigView.

There are six panes. Five of them are simple captions that we won't discuss any further. The pane in the lower left corner of the window is a custom CViewInfoPane. It reports cursor coordinates when the cursor is in the CBigView.

Examine the PPob resource with Constructor as you read the description of the views and panes.

Figure 7.13 The Views window



The LActiveScroller for the picture is bound to the top left of the window. Its scrolling view has an ID of 2. The LPicture view has an ID of 2. It is bound on all sides to its superview (the LActiveScroller). It is visible, but not enabled, so it will not receive clicks. The LPicture uses a PICT resource with ID 1000. This PICT has been provided for you in the application resources. The scroll unit for the LPicture is 5 pixels horizontally and vertically. A click in a scroll arrow scrolls the picture by 5 pixels in the correct direction.

The LScroller in the top right corner of the window is bound to the top, left, and right of the window. Its scrolling view has an ID of 4. The LTextEditView object has an ID of 4. It is bound on all sides to its superview, the LScroller. It is visible and enabled, so it will re-

ceive commands, keystrokes, or clicks (LTextEditView is a commander and a pane, so it receives all three). The initial text is in a TEXT resource with ID 1000, provided for you in the application resources. It uses the text traits resource with ID 130.

The LScroller in the center left of the window is bound to the top left of the window. Its scrolling view has an ID of 6. The LTable view has an ID of 6. It is bound on all sides to its superview, the LScroller. It is visible and enabled, so it will receive clicks. It has 5 columns and 20 rows. Notice that the scroll unit matches the size of a cell. One click in a scroll arrow will scroll the table one cell left, right, up, or down.

All of these are standard PowerPlant views.

The interface also includes a custom pane, CViewInfoPane. You built a custom pane in the previous exercise. We give you the pane in this exercise. It is bound to the top, left, and bottom of the window. Note the class ID is InfP. This pane is visible, but not enabled so it does not respond to clicks.

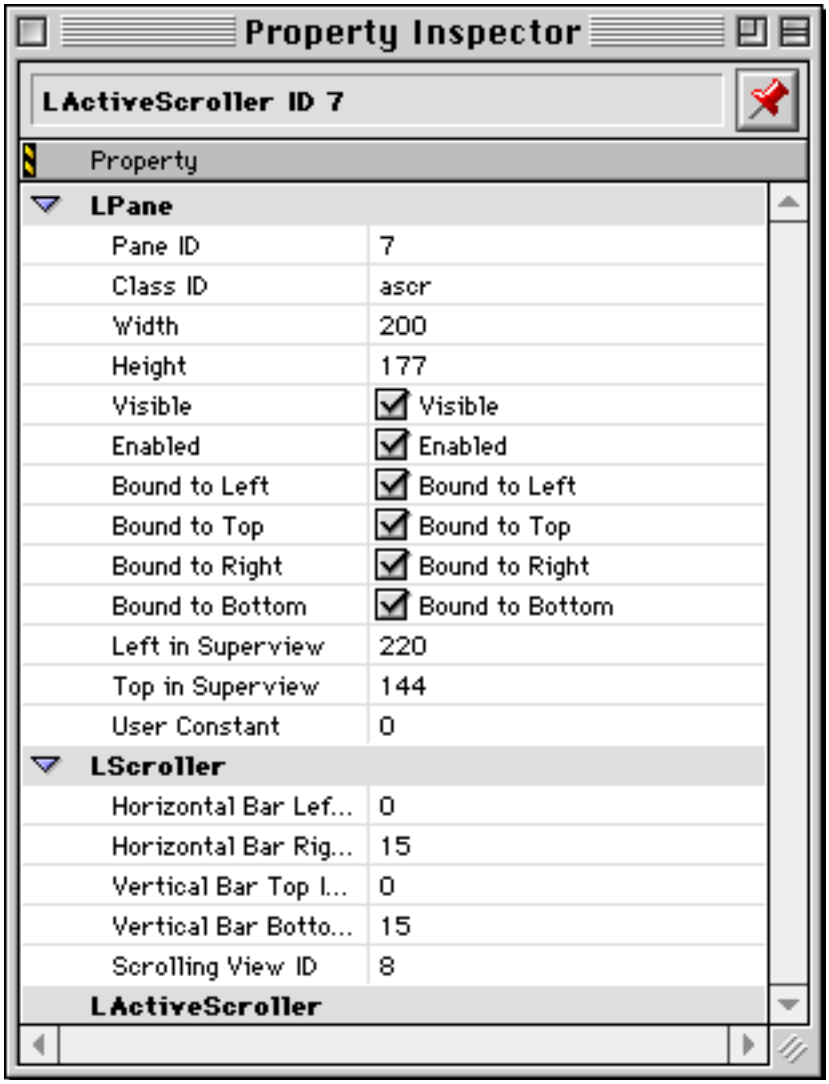
There are two views missing from the PPob, an LActiveScroller and the CBigView.

1. **Create an active scroller.**

   If you have not already opened the PPob project file, double-click the Views.ppob file in the IDE project window. Constructor launches and the Constructor project window appears. Double click the LWindow view to see its contents. Also, make sure the Catalog window is open.

   Drag an LActiveScroller object from the Catalog window and drop it on the Views window. Double-click the object to open the property inspector window so you can set its characteristics as shown in Figure 7.14.

Figure 7.14 Setting LActiveScroller properties



Set the location and size. The view is bound to all four sides of the window. It is enabled and visible. The Pane ID is 7, the Scrolling View ID is 8.

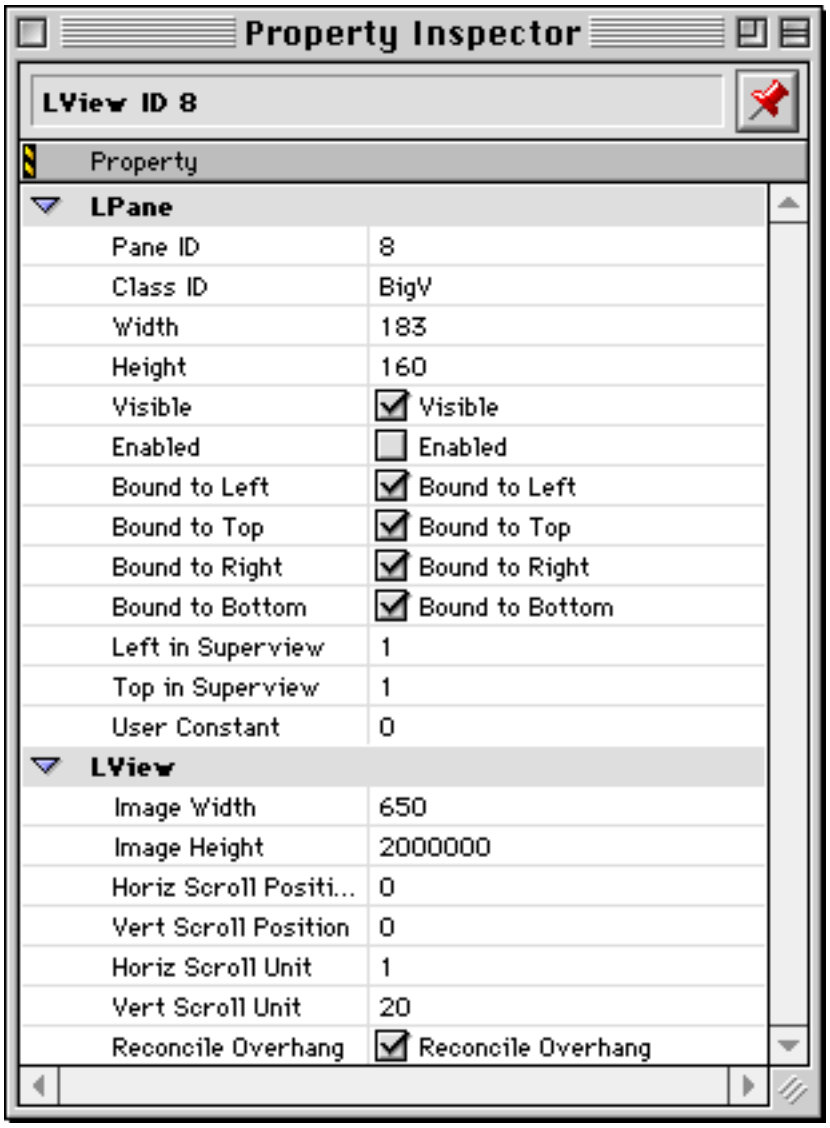
Close the LActiveScroller property inspector window.

2. **Create a custom view.**

CBigView is based on a plain LView object. Drag an LView object from the Catalog window and drop it inside the LActiveScroller

you just created in the Views window. Double-click the CBigView object to open the property inspector window so you can set its characteristics as shown in Figure 7.15.

Figure 7.15 Setting CBigView properties



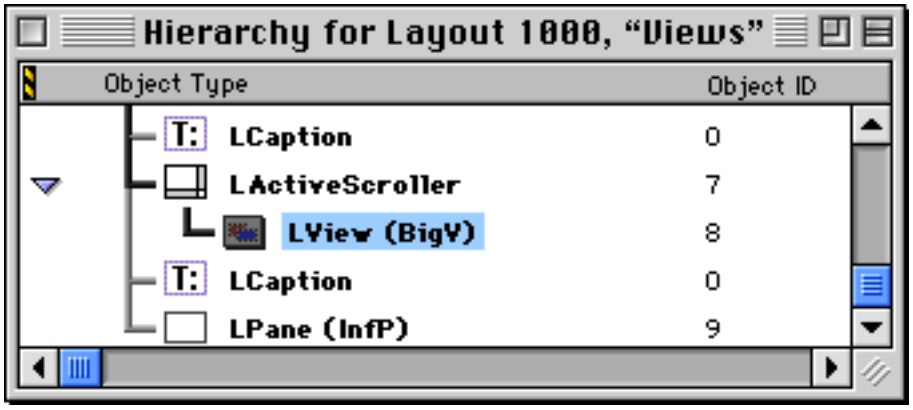
Set the location and size. The view is bound on all four sides to its superview. The view is visible but not enabled. The Pane ID is 8. The Class ID is BigV. Set the image size and scroll unit. The vertical

scroll unit is 20 pixels. This is an important value to guarantee that the solution code works correctly, so don't change it. Turn Reconcile Overhang on.

3. **Examine the view hierarchy.**

   If you dropped the CBigView inside the LActiveScroller, you should be all set. However, the hierarchy is critical so let's make sure. Open the Constructor hierarchy window. Make sure the new LView pane (representing the CBigView object) is hierarchically under the LActiveScroller. If it is not, reposition it so that it is. The end result should look like Figure 7.16.

Figure 7.16 The view hierarchy



Save your changes, close all the Constructor windows, and return to the CodeWarrior IDE. It's time to write some code.

Implementing a Custom View

All the views except for CBigView are standard PowerPlant objects. You don't have to write *any* code to make them work properly. The application-level code that creates the window has been provided for you. You'll work with applications in Chapter 9, and windows in Chapter 11.

In the remaining steps you write the code to implement the custom view.

4. **Declare the class ID.**

`Class declaration` `CBigView.h`

Most of the header has been provided for you. PowerPlant relies on each pane class (and a view is a kind of pane) having a unique class ID. In Step 2 you specified "BigV" as the class ID for the CBigView class.

Each view class has a `class_ID` enumerated constant. Declare that constant to have the value BigV.

```
class CBigView : public LView {
public:
  enum { class_ID = 'BigV' };
```

Examine the header briefly. The only function CBigView overrides is `DrawSelf()`. You write this function in the next few steps. All the code for the class creator function, the constructors and destructor has been provided for you. You wrote similar functions in the Panes exercise.

Save your changes and close the header file.

The `CBigView::DrawSelf()` function is fairly complex. You write this function in the next three steps. Follow the code carefully. What you're going to do in the next three steps is:

- Figure out what part of the image is visible.
- List the vertical image coordinate every 20 pixels.
- Draw a grey box halfway down the image.

The significant work you do with respect to PowerPlant is coordinate conversion. You must change back and forth from local to image coordinates and back again to get things to draw properly.

5. **Get the coordinates for the exposed image.**

`DrawSelf()` `CBigView.cp`

The existing code at the start of this routine preserves the text state of the port, then sets it to Geneva 9 point.

TIP: The StTextState class preserves and restores existing text settings. You can read about this class in "UDrawingState" on page 560.

After that you have three tasks to accomplish:

a. Get the view frame in local coordinates.

Define a `Rect` variable and call `CalcLocalFrameRect()`.

b. Convert the frame to image coordinates.

Define two `SPoint32` variables. Use these in calls to `LocalToImagePoint()`. Convert the top left of the frame and the bottom right of the frame from local to image coordinates.

c. Get the size of the image.

Define an `SDimension32` variable. Use the `GetImageSize()` accessor.

The necessary code for all these tasks is listed here.

```
TextSize( 9 );

// Calculate the frame rect in local coord.
Rect theFrame;
CalcLocalFrameRect( theFrame );

// Convert the frame to image coordinates.
SPoint32 theTopPos;
SPoint32 theBotPos;
LocalToImagePoint( topLeft(theFrame), theTopPos );
LocalToImagePoint( botRight(theFrame), theBotPos );

// Get the image size.
SDimension32 theImageSize;
GetImageSize( theImageSize );
```

Existing code then manipulates the value of the bottom image coordinate to allow for the height of the text you're going to draw, and

to make sure it stays within the image. This code is given to you because it has nothing to do with PowerPlant.

Save your work before proceeding.

6. Draw the vertical image coordinate string.

`DrawSelf()` `CBigView.cp`

In this step you write a loop. For every 20th pixel, you convert the vertical image coordinate into a string, and draw the string in the view. You plot a point in the view image, then convert it to local coordinates before drawing at that point.

a. Indent the horizontal drawing position.

Define an `SPoint32` variable. We'll call it `theImagePos` for reference. This point will hold the image coordinate at which you wish to draw.

Set the horizontal component of `theImagePos` to 4. You want the text to draw just a little in from the left edge of the view, for aesthetics.

b. From the top of the visible image, step every 20 pixels to the bottom of the visible image.

In Step 5 you created local variables to hold the top left coordinate of the visible part of the image, and the bottom right coordinate. Use the top and bottom values as the beginning and ending points of a `for` loop.

Set `theImagePos.v` to the top value. Then step by 20 pixels. Loop as long as `theImagePos.v` is less than the bottom plus 12 pixels. Add 12 pixels to the test to ensure that any partial lines of text appear along the bottom of the view.

c. Convert the image coordinate to local coordinates.

The variable named `theImagePos` now has both the horizontal and vertical component set properly, in image coordinates. Define a `Point` variable and call `ImageToLocalPoint()` to convert `theImagePos` to local coordinates.

d. Create a string of the vertical image coordinate.

Define a local `Str15` variable. Call the Toolbox routine `NumToString()`. Use `theImagePos`.v as the number.

e. Draw the string.

Call the Toolbox routine `MoveTo()` to move to the local coordinate. Call the Toolbox routine `DrawString()` to draw the string.

The solution code is listed here for reference. Existing code (in italic) is provided so you can locate where to place this code.

```
if ( theBotPos.v > theImageSize.height ) {
  theBotPos.v = theImageSize.height;
}

// Set the horizontal image coordinate
SPoint32 theImagePos;
theImagePos.h = 4;

// Step every 20 pixels.
for ( theImagePos.v = theTopPos.v;
      theImagePos.v < theBotPos.v+12;
      theImagePos.v += 20 ) {

  // Convert image into local coordinates.
  Point theLocalPos;
  ImageToLocalPoint( theImagePos, theLocalPos );

  // Create the string.
  Str15 theString;
  ::NumToString( theImagePos.v, theString );

  // Draw the vertical coordinate string.
  ::MoveTo( theLocalPos.h, theLocalPos.v );
  ::DrawString( theString );
}
```

Existing code then prepares for the next step, in which you draw a box in the image. Save your work.

7. Draw a box in the image.

`DrawSelf()` `CBigView.cp`

In this step you draw directly into a view with a 32-bit image. The steps you take are similar to those in the previous step.

Existing code sets up a box that is 30 pixels square. To draw this box in the view you must:

a. Define an image coordinate at which to draw.

You can use `theImagePos` variable again from Step 6. In Step 5 you got the size of the image. Set `theImagePos.h` to 100. Set `theImagePos.v` to one-half the image height.

b. Determine if any part of the box is visible.

Call `ImageRectIntersectsFrame()`. This call takes four 32-bit values representing the left, top, right, and bottom coordinates. Pass the correct values based on `theImagePos`. If the call returns true, part of the box is visible and it should be drawn.

c. Convert the coordinate to local coordinates.

Define a `Point` variable and call `ImageToLocalPoint()` to convert `theImagePos` to local coordinates.

d. Draw the box.

The box is at (0,0). It should appear at the local position you calculated in substep c above. Call `OffsetRect()` to reposition the rectangle. Then call `FillRect()`. The solution code uses `UQDGlobals::GetQDGlobals()` to access the gray pattern.

The solution code is listed here for reference. Existing code (in italic) is provided so you can locate where to place this code.

```
const Int16 kBoxSize = 30;
Rect theBox = {0,0,kBoxSize,kBoxSize};

// Where the top left of the box will be.
theImagePos.h = 100;
theImagePos.v = theImageSize.height / 2;

// Draw the box if some of it is in the frame.
if ( ImageRectIntersectsFrame( theImagePos.h, theImagePos.v,
```

```
        theImagePos.h + kBoxSize, theImagePos.v + kBoxSize ) ) {

  // Convert the image point to a local point.
  Point theLocalPos;
  ImageToLocalPoint( theImagePos, theLocalPos );

  // Now we can use QuickDraw to draw it.
  ::OffsetRect( &theBox, theLocalPos.h, theLocalPos.v );
  ::FillRect( &theBox, &UQDGlobals::GetQDGlobals()->gray );
}
```

Save your work and close the file.

8. **Register the new class.**

`CViewApp()` `CViewApp.cp`

You must register any custom PPob class. The application constructor already registers one custom class, the CViewInfoPane. We give you that class because it isn't directly related to views, and it does a few things you haven't learned about yet (like installing itself as a periodical task).

However, you must register the CBigView class you created. Enter the new code after the existing call to `RegisterClass_()`. The required header files are added for you.

```
//Register custom classes
RegisterClass_(CViewInfoPane);
RegisterClass_(CDemoTable);
RegisterClass_(CBigView);
```

9. **Build and run the application.**

Make the project and run it. When you do, a window should appear containing all the views. See Figure 7.13 on page 202. Play with the window and watch what happens.

Scroll each of the views, and watch what happens. Drag the thumb in the LScroller and the LActiveScroller views, and observe the difference in behavior. Notice how easy it is to implement scrolling.

None of the code you wrote had anything to do with scrolling in either type of scroller. It is a gift from PowerPlant.

Observe the LTable view. The numbers you see are the default behavior of the LTable class. In a real application you would replace that behavior to draw your own data of whatever type. The default LTable is not a commander, so it does not receive keystrokes. Typing an arrow key does not modify the selection. LTable also does not support multiple selection. New table classes are planned for PowerPlant that will implement some of these features.

The LTextEditView field is fully functional. LTextEditView supports all of the more advanced features you like to see in a text field, such as automatic scrolling.

Best of all, take a good look at CBigView. After all, you wrote the code for this one. Scroll the view. Observe the vertical coordinate lines as you do. Keep on going, with two million pixels you can go a long way. Use the thumb and drag to somewhere near the middle. The grey box is at the halfway point. When you created the view in Constructor, you set the image to be two million pixels high. Scroll until you reach the million-pixel coordinate to find the grey box. Not bad!

To observe the effect of "Reconcile Overhang," scroll the big view all the way down and to the right. Now resize the window larger. Observe what happens to the view in CBigView. The two-million pixel mark remains at the bottom of the scrolling view.

As you move the cursor in CBigView, observe the contents of CViewInfoPane. As long as the cursor is within CBigView, it displays the coordinate of the cursor in all four coordinate systems: global, port, local, and image. Observe the global and port coordinates. Move the window and look again, to see how they vary relative to each other. With CBigView scrolled fully up and to the left, observe that local and image coordinates are the same. Scroll down until they no longer match. When does that happen? Locate the grey box you drew, and put the cursor at the very top left corner of the box. What are the image coordinates?

When you are through observing, quit the application.

If you'd like to explore further, feel free to do so. Create new and different views. Change the values of the existing views in Con-

structor. Draw new items in the CBigView at various locations and practice coordinate conversions. Create your own views, with scrolling views inside. Experiment with LActiveScroller views and plain LScroller views.

You may want to explore CViewInfoPane as it uses a periodical to work its magic. We'll discuss periodicals in "Periodicals" on page 506. If you examine the `DrawSelf()` function for this pane, all it does is frame the pane. Where does all that data come from? The `SpendTime()` function draws it every time there is an idle event. It uses a utility class that is not officially part of PowerPlant, UDrawTypes. If you go exploring in here, check out `ShowViewPoints()` for some classic PowerPlant coordinate conversion. You should have a good time.

Congratulations! You have implemented several different kinds of standard PowerPlant views, as well as a completely new custom view with an image two million pixels high. You're making great progress.

There's only one more fundamental visual building block in PowerPlant, the control. And that's what we talk about next.

8

Controls and Messaging

This is the third and last chapter in the Basic Building Blocks section of the manual. In the previous two chapters we discussed panes and views. In this chapter we talk about the third and final group of pane classes, LControl and its descendants.

The principal topics in this chapter are:

- What Is a Control—including the different kinds of controls in PowerPlant.
- Control Characteristics—a detailed look at the things that make a control a control, including broadcasting messages.
- Working With Controls—how to make and use controls, and how to send and receive messages.
- Specific Control Classes—details on all the control classes.

After we complete this discussion, you'll create and manipulate real controls in this chapter's coding exercise.

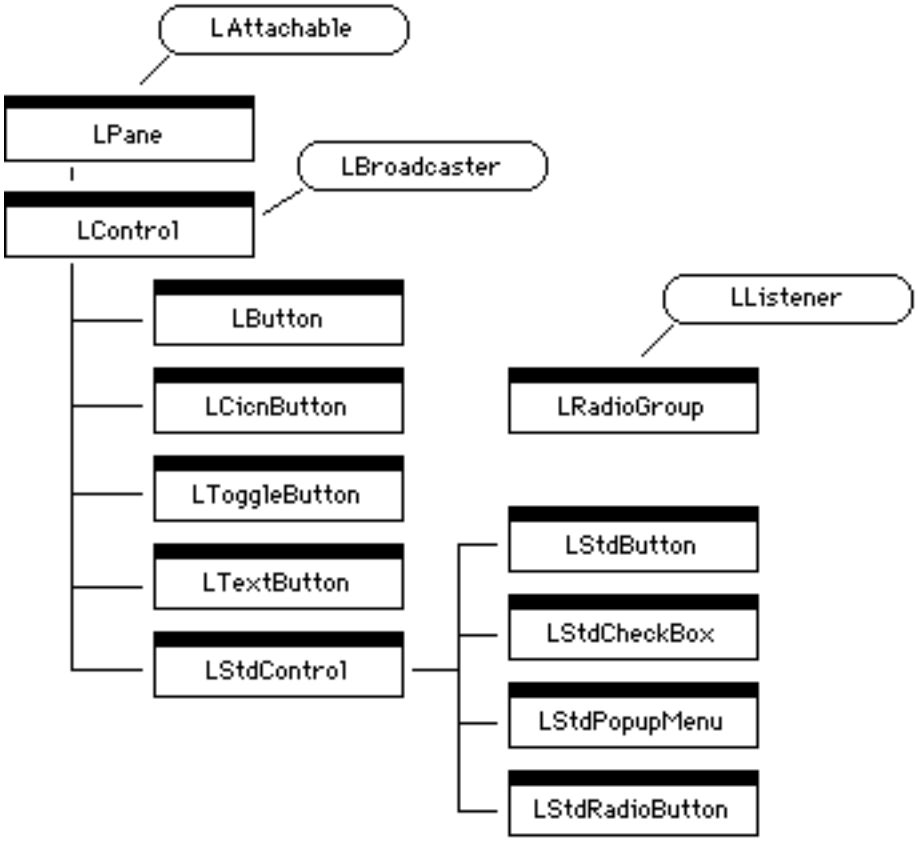
What Is a Control

From the point of view of the Mac OS, a control is a visual interface device with which the user controls the machine. Radio buttons, check boxes, popup menus, and so forth, are all controls.

From the PowerPlant point of view, a control is an object of any class that inherits from the LControl class. In the control classes, PowerPlant provides all of the standard controls you find in the Mac OS, and a few others as well.

Figure 8.1 shows the class hierarchy for the PowerPlant control classes.

Figure 8.1 Control class hierarchy



Notice that the LControl class inherits from LBroadcaster. This is the most significant difference between a control and any other kind of pane. All classes that descend from LControl are broadcasters—they can send messages.

TIP: Remember, for detailed information on any PowerPlant class, including a list of its ancestors, its member functions, and data members, you should refer to PowerPlant Revealed.

PowerPlant provides several extremely useful buttons: a plain button, a button that uses color icons, a text button, and a toggle button. We'll explain each of these particular button classes in "Specific Control Classes" on page 233.

PowerPlant provides four classes that implement standard control items in the Mac OS: the regular push button, a check box, a popup menu, and a radio button. All of these classes descend from LStdControl, which encapsulates standard, Mac OS control item behavior.

Finally, Figure 8.1 includes a related class that does not descend from LControl, the LRadioGroup class. LRadioGroup manages a group of mutually exclusive buttons, such as radio buttons, where one and only one button must be on at any given moment.

You may have noticed that LScroller and scroll bars are nowhere to be found in the LControl hierarchy. What gives?

Keep in mind that the LScroller class is not a control class. LScroller is a descendant of LView, and represents a scrolling view area. The LScroller class may contain two control objects—the two scroll bars. However, LScroller is not itself a control. The `LScroller::MakeScrollBars()` function creates these standard controls on the fly.

Although not obvious, scroll bars *are* included in the LControl hierarchy. Scroll bars are LStdControl objects. The LStdControl class (from which other standard Mac OS controls inherit), completely describes the behavior of a scroll bar.

Now that you know what the various kinds of controls are, let's talk about what makes these particular panes so useful.

See also "Managing Scrolling" on page 190 for more on LScroller.

Control Characteristics

Remember that all controls are also panes. Therefore, everything we said in Chapter 6, "Panes" applies to controls as well. They have an ID number, a frame, frame binding, contents, state, and mouse information. All of the data members and member functions discussed in that chapter apply equally to controls. Controls also use the value and descriptor characteristics of the LPane class.

In this section we discuss the additional features of controls that make them different from panes, and how controls use the value and descriptor.

The topics covered are:

- Control Values—a control's values, what they are and what they mean.
- Control Descriptor—how some controls use the descriptor characteristic of a pane.
- The Hot Spot—what it is, and how PowerPlant uses it.
- Broadcasting and Listening—how a control sends messages.

Control Values

A control uses the **value** characteristic of its pane nature to represent its current condition. For example, a radio button's value may be zero if it is off, or one if it is on. A scroll bar control has a range of possible values. Because the value may be a number in a range of values, each control also has a minimum and maximum value.

There is one more value of significance, the **message**. Because each control is a broadcaster, it has the ability to broadcast a message. LControl provides storage for the message to be broadcast (typically when the control is clicked). The message value is stored in a data member named `mValueMessage`. We'll talk about messaging in detail in "Broadcasting and Listening" on page 220.

Table 8.1 lists the value-related data members and their purpose.

Table 8.1 Control values

Type	Data Member	Purpose
`Int32`	`mValue`	current value for control
`Int32`	`mMinValue`	minimum control value
`Int32`	`mMaxValue`	maximum control value
`MessageT`	`mValueMessage`	message to be sent

NOTE: Do not confuse the `mValue` and the `mValueMessage` fields. Although similarly named, they are used for very different purposes. The `mValue` field is the current value of the control. The `mValueMessage` field is a number that is broadcast at appropriate moments, typically when a control is clicked. We will call this the message.

See also "Value and descriptor" on page 133.

Control Descriptor

Some controls have titles, in which case the title is stored as that control's descriptor. LControl itself does not provide direct support for descriptors. That is left to the individual control classes that need to use the descriptor.

The PowerPlant classes that have text titles are LTextButton, and LStdControl (and its descendants). LTextButton provides a data member to store the title, named `mText`. The LStdControl classes store the title in the Mac OS control record.

See also "Value and descriptor" on page 133.

The Hot Spot

A control's hot spot is the place where action happens when the user clicks. For many controls, the hot spot is the entire frame of the control. The various kinds of buttons are good examples of this type of control. No matter where in the button you click, you alter the button's state.

Other controls may have different parts that respond differently, and thus have a variety of hot spots. A scroll bar is a perfect example of a control with multiple hot spots. A scroll bar has two small arrows for scrolling line-by-line in opposite directions, a bar for scrolling page-by-page in opposite directions, and a movable thumb for setting the scroll position to some arbitrary location. A click in any of these five locations results in different behavior.

The good news is that, for all the PowerPlant classes, you don't have to worry about the hot spot. PowerPlant takes care of all the housekeeping details. However, the LControl class does provide a series of routines for managing the hot spot. We'll discuss those in "Managing the hot spot" on page 226.

Broadcasting and Listening

As we have already mentioned, the principal distinguishing characteristic of a control is that it is a broadcaster. What does that mean? On the code level, it means that LControl inherits from LBroadcaster. LBroadcaster is one of two messaging classes in PowerPlant. The other is its converse companion, LListener. LBroadcaster is a mix-in class used to add messaging capabilities to an object. LListener is a mix-in class used to allow objects to listen for messages from any arbitrary broadcaster.

Being a broadcaster, a control has the ability to broadcast a message. Each broadcaster (in this case each control) has a list of listeners, and the ability to modify the list on demand. When it broadcasts a message, the control sends the message to every one of its listeners.

In PowerPlant, a message has two components. One is a 32-bit integer of type `MessageT`. The other is a pointer to some data. For example, in a typical message from a control, the control broadcasts its message (the `mValueMessage` field), and a pointer to its current value (the `mValue` field). In this way any object listening to the message knows the current value of the control, and the nature of the message. Therefore the listener can respond appropriately.

In the regular PowerPlant classes, a control typically sends a message when it is clicked, or when its value changes. In your own control classes derived from the LControl hierarchy, you may send a message at any appropriate moment. We'll talk about how to send messages in "Broadcasting" on page 227.

NOTE: Although `mValueMessage` is typed as a `MessageT` data type, it is very common to have a message sent that is typed as a `CommandT`—a command that you want to be obeyed when the

message arrives at its destination. You'll encounter commands in Chapter 10.

In summary, the features that differentiate a control from other types of panes are:

- The use of additional values.
- The use of the descriptor.
- The hot spot.
- The ability to broadcast a message.

Now that you have a solid grasp of what makes a control a control, let's talk about how to use controls in PowerPlant.

Working With Controls

LControl is a fairly straightforward class in PowerPlant. Because a control is also a pane, you have already mastered most of the difficult concepts. In this section we talk about:

- Creating a Control—using Constructor, and constructor functions for control classes.
- Drawing a Control—drawing and updating controls.
- Managing Control Characteristics—managing the features of a control such as the message or the hot spot.

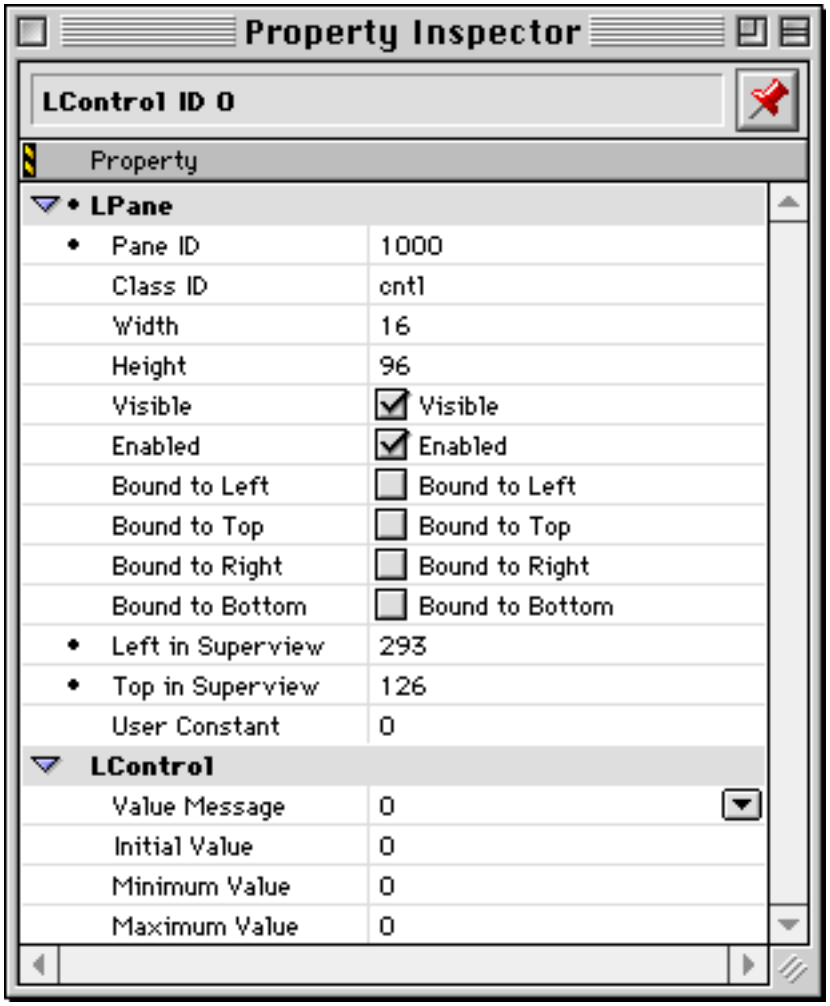
Creating a Control

You can create a control using Constructor, or on the fly in your code. We talk about each method. Then we discuss what you do when you derive your own class from LControl or its descendants.

Using Constructor

Creating a control object in Constructor is simple. While in Constructor, you drag a control object from the Catalog window into a containing view. Then you set the characteristics for that object, as shown in Figure 8.2.

Figure 8.2 Creating a control in Constructor



Note that a control has all the same information as a pane, including location, size, pane ID, class ID, binding, and state information. Remember, when you derive your own classes you must change the class ID to your own unique value and register the class with PowerPlant before creating any objects of that class.

Figure 8.2 is the Property Inspector window for an LControl object. You would typically use one of the specific controls such as LButton or LStdRadioButton. Each control object has its own properties. We'll discuss the individual controls in "Specific Control Classes" on page 233.

Remember, the general process for creating any control object with Constructor is: drag the item into the view, then set its characteristics.

See also "Register PowerPlant Classes" on page 288.

Creating a control on the fly

The typical approach used when creating a control object on the fly is to define an `SPaneInfo` structure to describe the pane-related features. You then call the appropriate constructor with the `SPaneInfo` and additional parameters specific to the type of control. The `LScroller::MakeScrollBars()` function is a good example of creating a control object on the fly.

Each control class has specific constructors of course. Refer to the *PowerPlant Revealed* HTML reference for details on the various constructors and the parameters you must provide to successfully create a control object on the fly.

After you have created a control and installed it in a view, you should call `FinishCreate()`. This function ensures that the pane's state (visible/invisible, active/inactive, enabled/disabled) matches its superview. It also calls `FinishCreateSelf()`. The `FinishCreateSelf()` function gives you the opportunity to provide "finishing touches" when creating a pane or view, because there may be times when you can't fully initialize a pane in its constructor.

If you are creating control objects on the fly, it is your responsibility to maintain the correct visual hierarchy. You must install your control in a view. If the control you are creating is or derives from LStdControl, there is one additional requirement. The current port must be the window into which you are installing the view. To ensure that it is, you can use code similar to that in Listing 8.1.

Listing 8.1 Establishing the port for an LStdControl

```
// install myControl into myView
myView->EstablishPort();
myControl->PutInside(myView);
```

See also "Creating a pane on the fly" on page 138 for more on the SPaneInfo structure.

Deriving your own controls

When you derive a class from LControl or one of its descendants, you typically define a destructor and several constructors: a default constructor, a constructor to build the control from an `SPaneInfo` structure and other parameters, a constructor to build the control from a stream, and a copy constructor. For more on stream constructors, see "Stream constructor" on page 141.

PowerPlant provides most of the controls you'll ever need in an application for the Mac OS. If you use custom controls, you'll have to derive classes to describe their behavior properly.

However, the most typical reason you would derive your own control class is because you want a control to listen to a message from some other control. For example, you may have a check box that, when turned on by the user, causes other controls to become enabled.

There are a couple of strategies you can use to accomplish this goal. You may have the view that contains the check box listen to a message, and then enable the necessary controls. A more direct approach (and one that doesn't require the view to know about its contents) is to make the dependent controls listeners to the check box. When the check box broadcasts the appropriate message, the dependent controls can enable themselves.

In order to make a control a listener, you must derive a class from some PowerPlant class in the LControl hierarchy, and mix in the LListener base class. We'll talk about how to use a listener in "Being a Good Listener" on page 231.

When you derive a new control class, you may override whatever functions are necessary. The functions you are likely to override include:

Table 8.2 Commonly overridden control functions

Function	Purpose
`ClickSelf()`	respond to a mouse click
`DrawSelf()`	draw the pane contents
`PointIsInFrame()`	determine if a click is in a hot spot
TrackHotSpot()	track the mouse when clicked in a hot spot
`HotSpotAction()`	act while the mouse remains down in the hot spot
`HotSpotResult()`	act when the mouse button is released in the hot spot

The `HotSpotAction()` and `HotSpotResult()` functions are empty in LControl, but defined in subclasses of LControl. You may also need to override the value and descriptor accessors if your control class uses those features in non-standard ways.

See also "Creating a control on the fly" on page 223 for more on the SPaneInfo structure.

Drawing a Control

Controls are just another form of pane, so drawing works the same way as it does for panes. In most cases, the default behavior provided in PowerPlant will suffice for your needs. `Draw()` performs housekeeping details and calls the control's `DrawSelf()` function.

You may override the `DrawSelf()` function if necessary to do drawing specific for your own control classes. For example, the LCicnButton class uses the Mac Toolbox call `PlotCIcon()` to draw itself.

Managing Control Characteristics

In this section we talk about the control-related functions you use to manage values, the descriptor, and the hot spot.

Managing control values

Control objects have accessor functions to manipulate the contents of the various values in a control. Table 8.3 lists the functions.

Table 8.3 Control value management functions

Function	Purpose
`GetValue()`	return the current value
`SetValue()`	set the current value
`IncrementValue()`	change the current value by the amount specified (could be negative)
`GetMinValue()`	return the minimum value
`SetMinValue()`	set the minimum value
`GetMaxValue()`	return the maximum value
`SetMaxValue()`	set the maximum value
`GetValueMessage()`	return the message
`SetValueMessage()`	set the message

Managing the control descriptor

For those control classes that use the descriptor feature, you have the same accessors available as for any pane. Predictably, they are `GetDescriptor()` and `SetDescriptor()`.

Managing the hot spot

This may be the trickiest part of managing a control. Happily, if you stick with the standard PowerPlant classes, you can forget all about it. If, however, you want to create custom controls, you should be aware of the control functions that relate to hot spots. Table 8.4 lists the functions.

Table 8.4 Hot spot management functions

Function	Purpose
`FindHotSpot()`	determine the number of the hot spot clicked
`PointInHotSpot()`	determine if a point is in a given hot spot
`TrackHotSpot()`	perform mouse tracking in a hot spot
`HotSpotAction()`	act while mouse is down in a hot spot
`HotSpotResult()`	act after mouse is released in a hot spot
`ClickSelf()`	what to do in response to a click
`SimulateHotSpotClick()`	make the control behave as if a hot spot was clicked, typically for key equivalents

Examine the PowerPlant classes to see how they implement these functions. For example, the `PointInHotSpot()` routine typically calls `PointIsInFrame()` to do the real work, so consider that routine as well when implementing custom control behavior.

Broadcasting

In this section and the next section we discuss the PowerPlant messaging system of broadcasters and listeners. In this section we accomplish two goals. We talk about broadcasting in general, and how control objects broadcast.

LBroadcaster

LBroadcaster is a simple class. It has only a few functions, and they are easily understood. Table 8.5 lists all the member functions except constructors and the destructor.

Table 8.5 Some LBroadcaster member functions

Function	Purpose
`AddListener()`	add a new listener
`RemoveListener()`	remove a listener
`StartBroadcasting()`	enable broadcasting
`StopBroadcasting()`	disable broadcasting
`IsBroadcasting()`	return current broadcasting state
`BroadcastMessage()`	send a message to each listener

That's all there is to LBroadcaster. However, hidden inside this simplicity is great power. The basic operations are simple. You can add or remove listeners, start or stop broadcasting, and broadcast a message.

Of course, any broadcast implies that someone is listening at the other end. You must explicitly connect listeners to broadcasters.

Linking broadcasters to listeners

Whenever you have an individual listener that you want to connect to a broadcaster, you simply call the *broadcaster's* `AddListener()` function.

This can be a trifle tedious when you have a single listener that listens to several control items. For example, a typical dialog box (LDialogBox inherits from LListener) may contain many control items to which it listens.

PowerPlant has a mechanism you can use to connect a listener to a set of controls, `UReanimator::LinkListenerToControls()`.

You pass three parameters to this function. The first is a pointer to the listener object you want to attach to the controls. The second is a pointer to the view object that contains all the controls. (All controls must be in a single visual hierarchy). Finally, you pass the resource

ID number of a special kind of PowerPlant resource, the RidL resource.

RidL stands for Resource ID List. A RidL resource is a list of one or more LControl or LControl inherited classes pane IDs.

Constructor creates a RidL resource automatically for every window that contains controls. The ID number of the RidL matches the ID number of the corresponding PPob resource created for the window. The RidL resource lists every control object in the window with a non-zero pane ID.

WARNING! If you use Constructor to edit a window or view that has no controls, Constructor deletes any RidL with the matching ID. If you edit a window with controls, Constructor replaces any existing RidL with the default list of all controls in the window.

You cannot see or edit the RidL in Constructor. If you wish to make a custom RidL resource, or edit an existing resource, you must use ResEdit or Resorcerer. You can use custom RidL resources to link a listener to an arbitrary set of controls, as long as all controls are in a single view hierarchy.

So, you have two methods of linking a listener to a set of control objects. You can use `AddListener()` for each control. Or you can use `LinkListenerToControls()` to link a listener to multiple controls in a single view hierarchy.

What about the converse situation, where you want to add a series of listeners to a single control? There is no batch method for accomplishing this task. You call `AddListener()` separately for each listener. However, a control with multiple listeners is an uncommon situation. Most broadcasters have very few listeners.

See also "RidL Resource" on page 116 for more on the RidL resource.

Broadcasting a message

You can turn a broadcaster on or off using the LBroadcaster functions `StartBroadcasting()` and `StopBroadcasting()`. You inquire about the state with `IsBroadcasting()`. If broadcasting is off, calling `BroadcastMessage()` has no effect.

Assuming that broadcasting is on, when you want the broadcaster to send a message you call `BroadcastMessage()`. You should never need to override this function. `BroadcastMessage()` walks through the broadcaster's list of listeners, and calls each listener's `ListenToMessage()` function. The broadcaster sends two parameters: a 32-bit number of type `MessageT`, and a void pointer.

Typically the first parameter is a value defined either by you or PowerPlant that describes the nature of the message, or the nature of the broadcaster. For example, PowerPlant defines the value `msg_ControlClicked` and sends that message when certain controls are clicked. You can send constants of type `CommandT` as well as MessageT, so you can broadcast commands directly to listeners.

The second parameter is the void pointer. It may be nil, or point to data or an object. You are free to pass whatever associated information is necessary for the listener to appropriately respond to your message.

See also the topic `PP_Messages.h` in *PowerPlant Revealed* for a list of defined commands and messages.

How controls broadcast a message

Controls have a separate broadcasting function that goes by the name `LControl::BroadcastValueMessage()`. Although related to the messaging system, this is not inherited from LBroadcaster. This is a function unique to controls.

`BroadcastValueMessage()` is a wrapper function for `LBroadcaster::BroadcastMessage()`. The typical control sends two values to its listeners. The first is the control's `mValueMessage` data member (the message), which contains a value you define. The second parameter is the current value of the control—the `mValue` data

member. The `BroadcastValueMessage()` simply calls `BroadcastMessage()` with these parameters.

However, there is nothing in this arrangement that prevents a control from calling `BroadcastMessage()` directly. Several PowerPlant control classes do just that. For example, the `LStdRadioButton::SetValue()` function calls `BroadcastMessage()` to pass the `msg_ControlClicked` and a pointer to the radio button.

Clearly, then, the PowerPlant messaging system is extremely powerful. You can modify the listener list at any time, send any message you want, and send the listener a pointer to any data. However, for all this to work someone must listen for and respond to the messages.

Being a Good Listener

Although PowerPlant's control classes do not inherit from LListener, this is a good place to discuss listeners in general. For one thing, listening is the other half of broadcasting. You can't have one without the other.

Secondly, it is a fairly common practice to derive your own control classes and have them inherit from LListener. Such a control is both a broadcaster and a listener.

LListener

The LListener class, like LBroadcaster, is a mix-in class. Use it to add listening capabilities to an object. Again like LBroadcaster, LListener is a very simple class. Table 8.6 lists the important functions of a listener.

Table 8.6 Some LListener member functions

Function	Purpose
`StartListening()`	start listening to broadcasters
`StopListening()`	stop listening to broadcasters

Function	Purpose
`IsListening()`	returns listening state
`ListenToMessage()`	respond to a message

LListener is an abstract class. You cannot create an LListener instance. You must inherit from it and override the `ListenToMessage()` function.

Linking listeners to broadcasters

All broadcaster-listener links should be made through the broadcaster, not the listener. Use `LBroadcaster::AddListener()` or `UReanimator::LinkListenerToControls()`, as discussed in "Linking broadcasters to listeners" on page 228.

You may have noticed that Table 8.6 does not list any functions that manage the link between a listener and a given broadcaster. Such functions do exist, but they are private member functions that you should never call directly.

Listening to a message

The `ListenToMessage()` function is the important part of any listener, and the only function you are likely to override. This function is typically called by broadcasters, but you can call `ListenToMessage()` directly at any time if you want a listener to respond to some message. This will ignore the listener's on/off state however.

The typical `ListenToMessage()` function identifies the nature of a message and then responds accordingly. If the listener may receive a variety of messages, you typically see a `switch` statement or some other flow-control mechanism that branches to the code designed to respond to each message. `LRadioGroup::ListenToMessage()` is a good example. Listing 8.2 shows how that function controls the flow based on the received message.

Listing 8.2 Excerpt from LRadioGroup::ListenToMessage()

```
switch (inMessage) {

  case msg_BroadcasterDied:
    …
    break;

  case msg_ControlClicked:
    …
    break;
}
```

When you declare a class that inherits from LListener, you must define the `ListenToMessage()` function. You decide what messages the listener should respond to, and write the code to handle the message. If you don't want to handle a message, you can just ignore it. There is nothing in PowerPlant that says you must respond to every message received.

Specific Control Classes

Let's close our discussion of control objects with a quick look at the various control classes and at the LRadioGroup class. The specific classes we discuss are:

- LButton
- LCicnButton
- LTextButton
- LToggleButton
- LStdControl
- LStdButton
- LStdCheckBox
- LStdPopupMenu
- LStdRadioButton
- LRadioGroup

Note that the LStdControl classes use the Mac OS Control Manager to accomplish much of their work. The other control classes do not.

LButton

LButton is a button that uses a graphical element as the visual representation of the button. The graphic is stored as a resource. The resource type may be one of these three options:

- ICN#—icon family
- ICON—black and white icon
- PICT—a picture

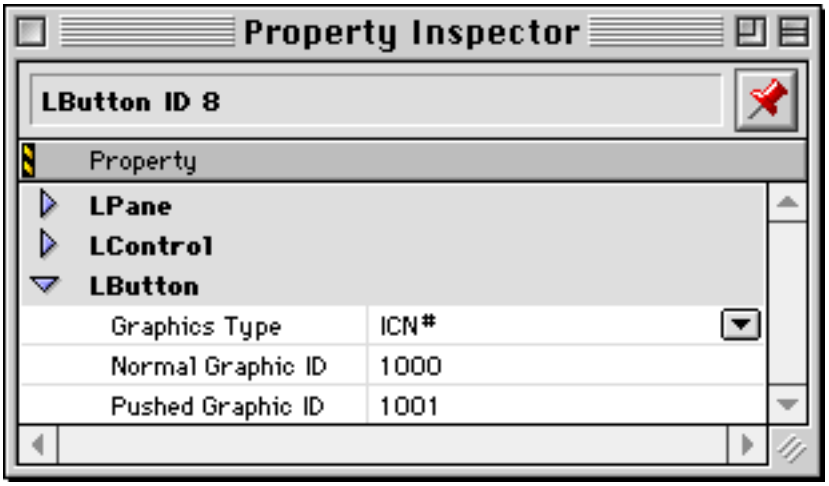
When you create the object, you provide two resource ID numbers. One is for the graphical element in its "normal" or non-pushed state. The other is for the graphical element you want to use when the user clicks or "pushes" the button.

If you use the ICN# resource as the basis for your graphical button, the Mac OS automatically picks the icon family member that best matches the display settings of the monitor on which it appears.

Although typical buttons are fairly small, and both ICON and ICN# resources describe images with specific dimensions, you can use a large PICT as a button.

Figure 8.3 displays the LButton-specific options you encounter in Constructor.

Figure 8.3 LButton properties in Constructor



You also set the usual pane and control characteristics such as the message, minimum, maximum, and initial values, and so forth.

When the user releases the mouse within the button, the button sends a message to its listeners. The value message depends on your application.

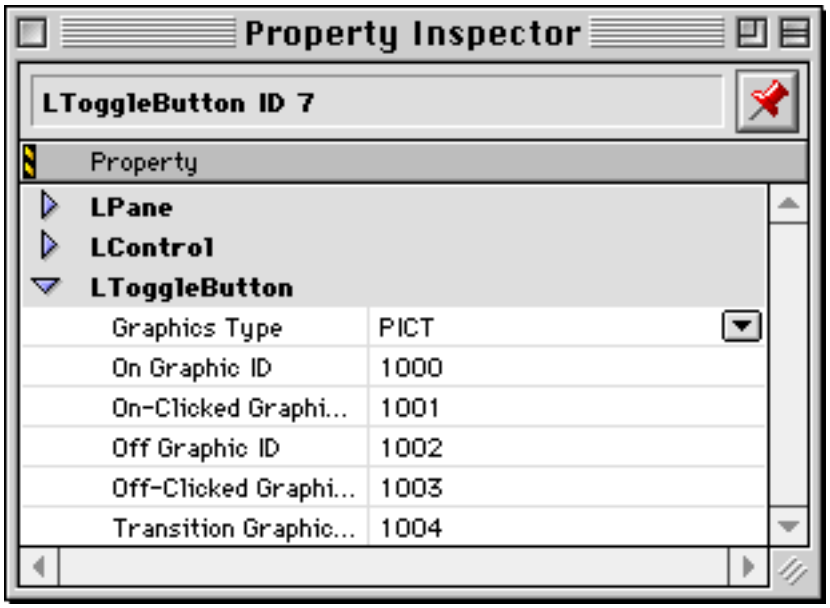
LCicnButton

This class is effectively identical to LButton, with one exception. It uses the Mac OS 'cicn' resource format as the graphical element. As with the LButton class, you provide resource ID numbers for the button in both normal and pushed state.

LToggleButton

The LToggleButton class is similar to LButton. It represents a graphical button using ICON, ICN#, or PICT graphics. Rather than being limited to "on" and "off" graphics, you may specify five separate resource ID numbers. Figure 8.4 illustrates the options.

Figure 8.4 LToggleButton properties in Constructor



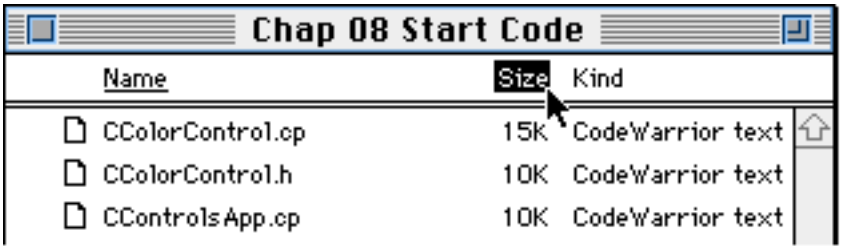
In addition to the regular on and off graphics, you have graphics for a click on a button that is already on, a click on a button that is already off, and a transition graphic that displays when the button is switching states.

You can use an LToggleButton to create some simple but intriguing animations that display when the user clicks a button. For example, you could create a drop-down flag button, or a door that opens and closes.

LTextButton

LTextButton describes a button with textual rather than graphical content. There is no standard Mac OS control item that matches LTextButton's behavior. However, you've seen similar controls in action. Figure 8.5 shows how the Finder uses a similar type of control as column titles in its list views.

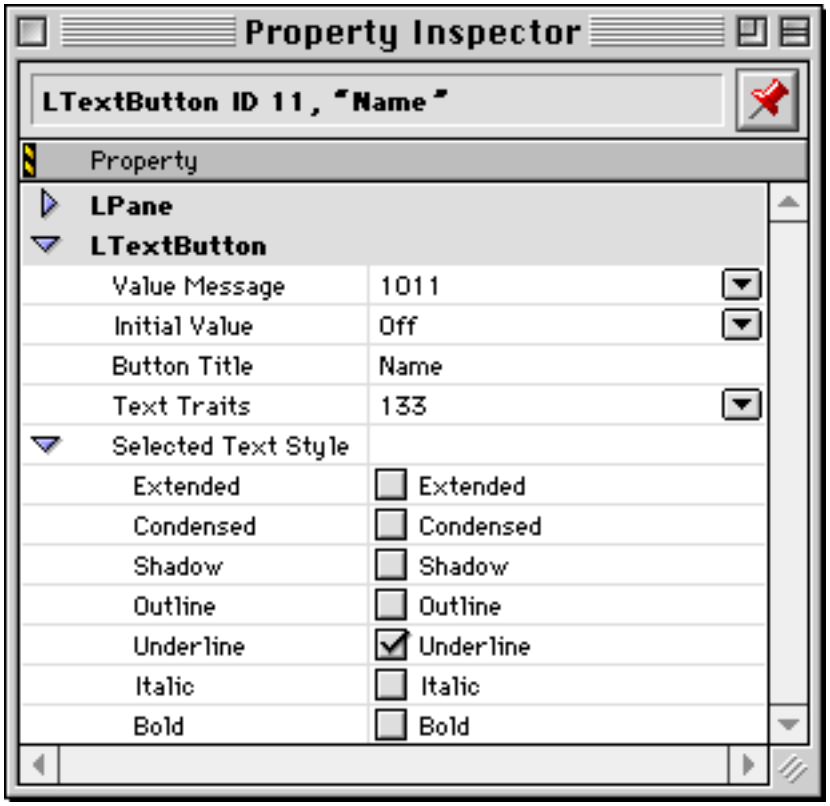
Figure 8.5 Text buttons in action



In Figure 8.5, the column titles are in fact text buttons that control sort order for the items in the window. The current sort order is by name—the "on" button appears in underline style. The user is clicking on the Size button and is about to change the sort order.

Figure 8.6 shows the specific LTextButton characteristics you set using Constructor.

Figure 8.6 LTextButton properties in Constructor



Like all PowerPlant text-related objects, you specify the style of text in a text traits resource.

Clicking an LTextButton toggles the button's state between on and off, like a radio button. The different state is represented visually by a change in the text style. The button style may become underline, bold, italic, outline, shadow, condensed, extended, or any combination of those style options.

Like other buttons, when the user clicks on an LTextButton the button sends a message to its listeners. In a typical scenario, you want some action to occur immediately when the user clicks a text button—for example, resorting the contents of a window. In that case, you should specify a value message that uniquely identifies the button to any listener. That way you'll know what button was clicked.

A series of LTextButton objects typically represents a set of mutually exclusive options. If that's how you use text buttons, then you should use the LRadioGroup class to ensure that one and only one button in the group of buttons is on.

See also "LRadioGroup" on page 245.

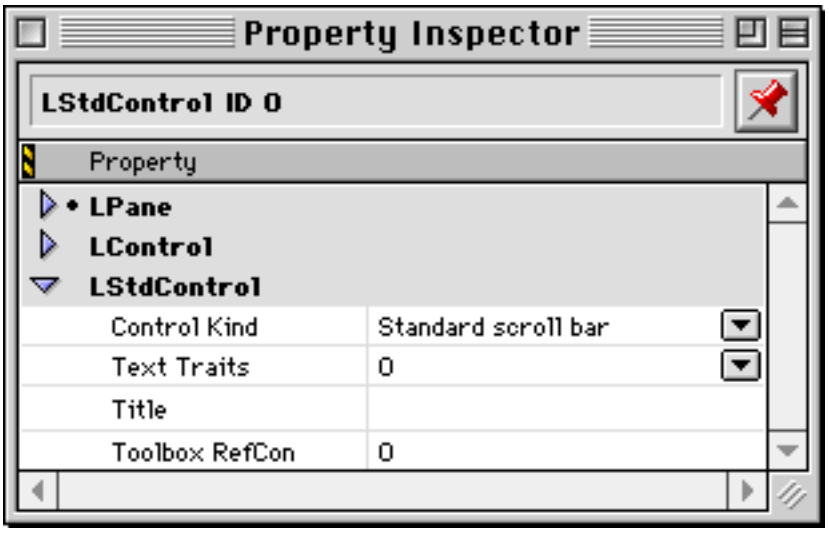
LStdControl

This class encapsulates the standard behavior of a Mac OS control item. The specific design of this class supports scroll bar controls and controls that use custom control definitions (CDEFs). This class also forms the basis for the standard button, check box, popup menu, and radio button classes.

NOTE: The `LStdControl.cp` file defines the functions for **all** the standard controls—radio buttons, check boxes, etc.

Figure 8.7 illustrates the characteristics you set for a standard control using Constructor.

Figure 8.7 LStdControl properties in Constructor

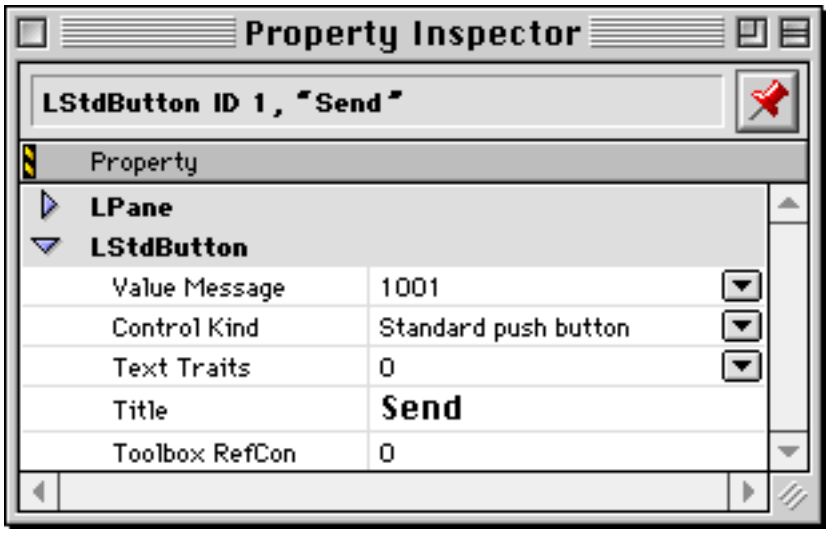


If you create a custom control with its own control definition function (CDEF), you specify the CDEF in the Control Kind field.

LStdButton

This class is PowerPlant's implementation of standard Mac OS push button behavior. You specify the text to display, a text traits resource ID, and the message to send when clicked.

Figure 8.8 LStdButton properties in Constructor



You do not need to specify a minimum or maximum value, because buttons have no real value. They simply highlight and send a message when clicked.

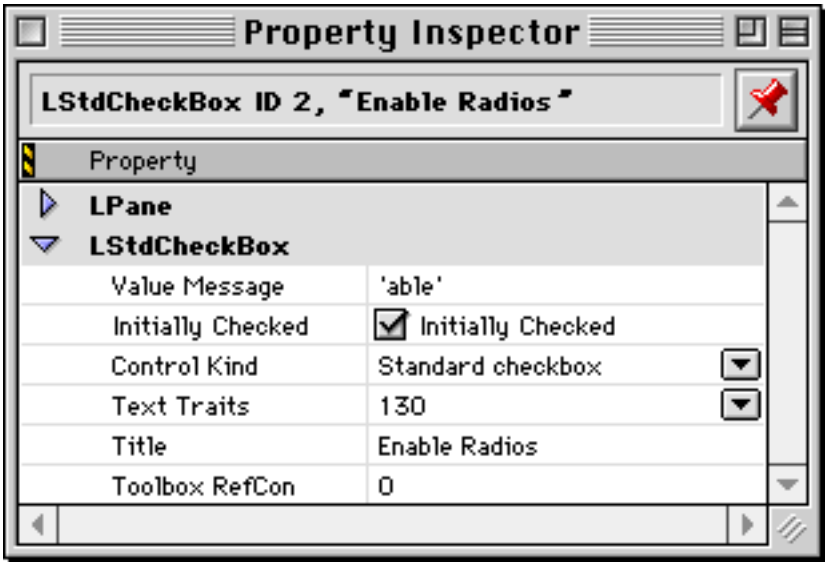
LStdButton does not draw the default-button outline. That detail is typically handled transparently for you by PowerPlant. View classes that have a default button—such as LDialogBox—let you specify a default button. PowerPlant uses the LDefaultOutline class to manage the process.

See also "LDefaultOutline" on page 151.

LStdCheckBox

This class is PowerPlant's implementation of a standard check box. It is a very simple class. The only function of LStdControl that it overrides is `HotSpotResult()`. If the user clicks this control, the control's value toggles between zero and one. Figure 8.9 displays the characteristics of an LStdCheckBox.

Figure 8.9 LStdCheckBox properties in Constructor

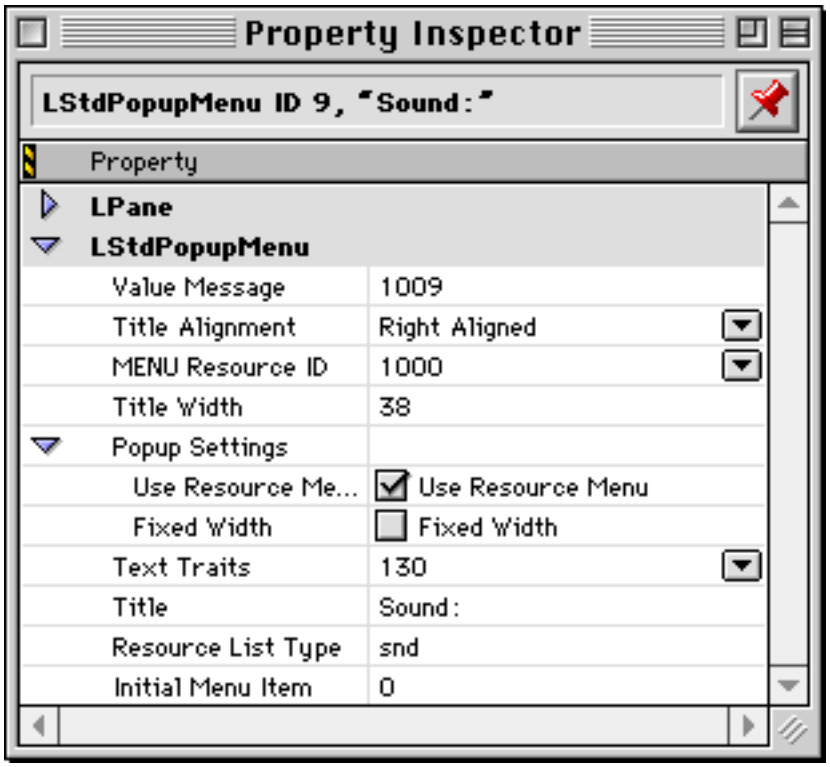


LStdPopupMenu

This class is PowerPlant's implementation of the standard Mac OS popup menu. Once again, the PowerPlant wrapper hides virtually every detail of the Mac OS from your view, freeing you to concentrate on real coding problems.

Figure 8.10 shows you the many characteristics specific to a popup menu that you can set using Constructor.

Figure 8.10 LStdPopupMenu properties in Constructor



Like many controls, you specify a title. The text traits resource ID applies to the entire menu—the items in the menu as well as the title. To specify unique style features for the title, use the check boxes in the Title Style group.

The title width is subtracted from the full width of the frame. If the Fixed Width option is on, menu items display in the remaining space. If it is off, the width of the menu is adjusted to allow the longest item to fit (regardless of the frame you set).

Title Placement controls the justification of the title text within the title space. The menu title is always to the left of the menu items. Within its allotted space the title may be flush left, centered, or flush right.

You specify the initial menu item that appears when the popup is displayed.

The Value Message typically identifies the popup menu itself. You might use the pane ID as the message, for example. When the user chooses an item in the menu, the menu broadcasts a message consisting of the value message (whatever you set), and the popup menu's current value. The current value is the item currently displayed, so you know what item the user chose.

You specify the resource ID of a MENU resource to set the contents of the menu. If you want the menu to be a list of resources, you click the Resource List check box and specify the resource type. You might specify 'snd ' for a menu of available sound resources, FONT for a font menu, and so forth. PowerPlant builds the menu contents for you automatically.

WARNING! If you modify the number of items in a popup menu at runtime, the LStdPopupMenu object does not automatically adjust its max value to represent the new number of items. You must do that manually.

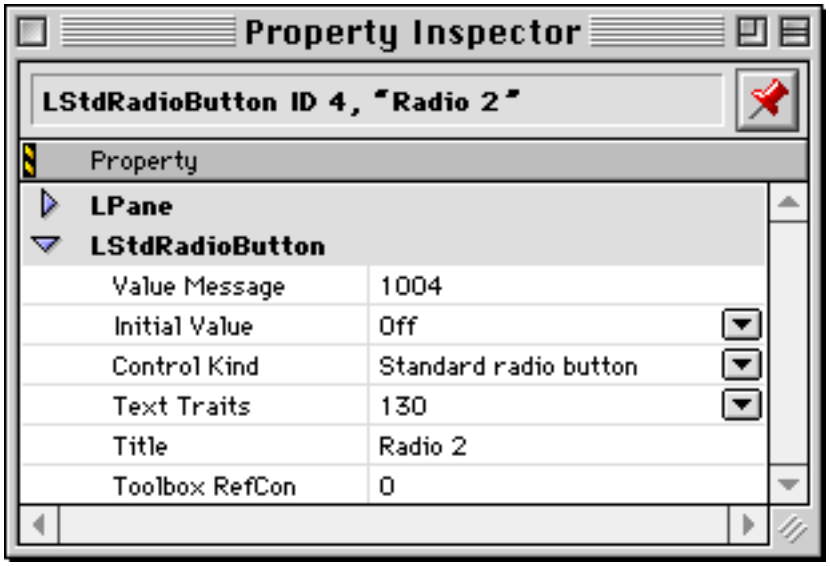
If you use the same menu in more than one place simultaneously and change the menu, the change affects all LStdPopupMenu objects that share the same MENU resource. Remember to update all objects as necessary.

LStdRadioButton

This class is PowerPlant's implementation of the standard Mac OS radio button. Like the LStdCheckBox class, this is a very simple extension of the LStdControl class.

A click on a radio button sets its value to one. The button then broadcasts a message that it has been clicked. The broadcast message also includes a pointer to the clicked button.

Figure 8.11 LStdRadioButton properties in Constructor



You specify a title, text traits resource ID, the value message, and an initial value.

The default behavior for LStdRadioButton does not use the value message. If you subclass from LStdRadioButton and use the value message, make sure you set its value appropriately.

In its default behavior, LStdRadioButton calls `BroadcastMessage()` directly with two parameters, `msg_ControlClicked` and a pointer to the radio button object. A listener can use the pointer to find out anything it needs to know about the clicked button.

If you want a listener to respond to the button *immediately*, you can use the LStdRadioButton pointer you receive in the message to get the pane ID or any other information you need, and then act accordingly.

You typically use radio buttons to describe a set of mutually exclusive options. Typically you set one button to have a value of On. If all the buttons are off in a group, PowerPlant will turn on the first button it encounters. If two or more buttons in a group are on, PowerPlant will turn off all but the last on button.

The LStdRadioButton class itself has no feature for grouping radio buttons, or for turning off other radio buttons in a group when a new button is turned on. That detail is handled by LRadioGroup.

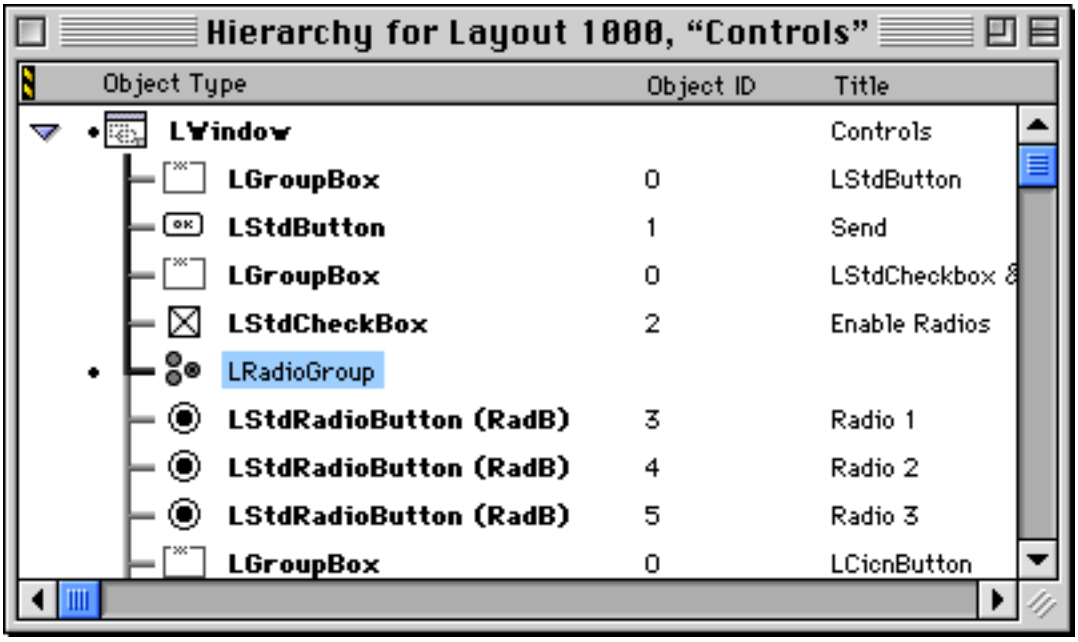
TIP: You can use the LRadioGroup class to manage groups of LTextButtons as well as LStdRadioButtons. A group of either kind of button represents a mutually exclusive set of options.

LRadioGroup

To create an LRadioGroup in Constructor, you first select a group of radio buttons. Then you choose **Make Radio Group** from Constructor's **Arrange** menu.

No visible item appears in the layout window, but you can see the radio group in the hierarchy window, as shown at the bottom of Figure 8.12.

Figure 8.12 Radio group in the hierarchy window



Membership in the radio group is based on the pane ID number of each radio button.

NOTE: The position of the radio group within the hierarchy is not important. It can come before or after the buttons in the group. However, if you rearrange the position of a radio button in the object hierarchy, the radio group will lose track of the button.

LRadioGroup works with LStdRadioButton and LTextButton. To use another kind of button with LRadioGroup (LToggleButton for example), override the button's `SetValue()` function so that the button broadcasts the `msg_ControlClicked` message. Use `LStdRadioButton::SetValue()` as an example. If the button does not broadcast this message, it will not be mutually exclusive.

Summary

Once again, you have just consumed a tremendous amount of information about PowerPlant. In this chapter you learned all about control objects and the PowerPlant messaging system.

Controls give the user choices for controlling the behavior of your application. PowerPlant supports about 10 different kinds of controls, including standard Mac OS controls and custom controls.

A control is a special kind of pane that is also a broadcaster. A control uses several additional values to keep track of its status, as well as the descriptor feature of the pane to manage the control title. Controls have "hot spots" and respond when the user clicks the hot spot. In most cases, a control has a single hot spot whose dimensions are identical to the control frame.

You learned how to create and display a control, how to manage a control's various characteristics, and how to respond to a click in a control.

In response to a click, a control broadcasts a message. You learned about the features of both LBroadcaster and LListener—what they

do, how they work, and how to link one with the other. You also learned how controls manage broadcasting.

Finally, you learned about each specific control class. You saw examples of typical uses, and studied the kinds of features each class adds to the base control classes in PowerPlant.

Now let's put that knowledge to work and create real controls in PowerPlant.

Code Exercise

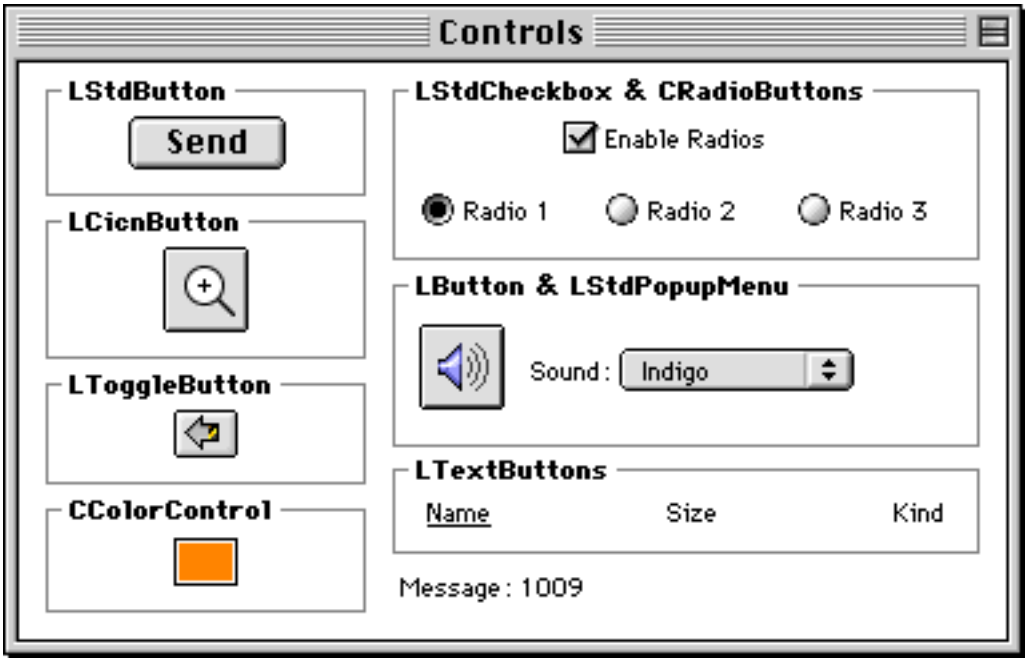
In this exercise you build an application titled "Controls." A PPob containing most of the controls is provided for you. You complete the interface, and then write the code necessary to make the controls work. This code exercise has three sections in which you implement

- The Interface
- CColorControl
- The Controls Application

The Interface

The final application looks like Figure 8.13, below.

Figure 8.13 The Controls window



When you click on a control, it sends a message to its listeners. This window displays the message. Several of the controls perform actual functions as well. For example, the button next to the sound popup menu plays the selected sound. The CColorControl in the bottom left corner changes colors.

There are thirteen controls in this window. Open the `Controls.ppob` file and examine the control characteristics with Constructor as you read the following descriptions.

None of the controls are bound to the superview—the window. This window does not resize, so binding is not an issue.

The LStdButton is a standard PowerPlant object. It has a value message of 1001. It uses no text traits resource, which means it uses the System font.

The LStdCheckBox is a standard PowerPlant object. It has a value message of "able"—a text message. It uses text traits resource 130.

The three radio buttons are custom controls. They are standard radio buttons that are also listeners. They listen to the check box so they can enable or disable themselves as necessary. They have value messages of 1003, 1004, and 1005 respectively. Notice that the class ID is RadB. You have already built a custom pane and view, so you know the importance of the class ID.

Open the Constructor hierarchy window, and look for an LRadioGroup object. In fact, you'll find two. One LRadioGroup is for objects 3, 4, and 5—these three radio buttons. The other is for objects 11, 12, and 13. These are the LTextButton objects. We'll talk about them in just a bit. The LRadioGroup button makes sure that one and only one control of its set of controls is on.

The LCicnButton is a regular PowerPlant object. It has a value message of 1006. It uses two cicn resources, ID number 1000 for its normal state and ID number 1001 for pushed state. These icons are provided for you in the application resources.

The LToggleButton is another regular PowerPlant object. It has a value message of 1007. It uses a series of five PICTs for the button image, numbered 1000-1004. These PICTs are provided for you in the application resources.

The LButton and LStdPopupMenu objects are regular PowerPlant controls. The LButton uses an icon family with resource ID 1000, provided for you in the application resources. It has a value message of 1008. The LStdPopupMenu uses MENU ID 1000, and text traits resource 130. It has a value message of 1009.

The three LTextButton objects are regular PowerPlant controls. The value messages are 1011-1013 respectively. Each uses text traits resource 130. There is an LRadioGroup object to control these buttons.

The CColorControl is a custom control derived from LControl. It also stores additional data, so it is a "custom type" in Constructor. You'll build this object in a little bit. It does not exist in the PPob in the start code.

Finally, there are 7 LGroupBox objects that we discuss no further, and two captions. One caption says "Message." The other caption is

blank. You'll write code to display the message received by the listener in this blank caption.

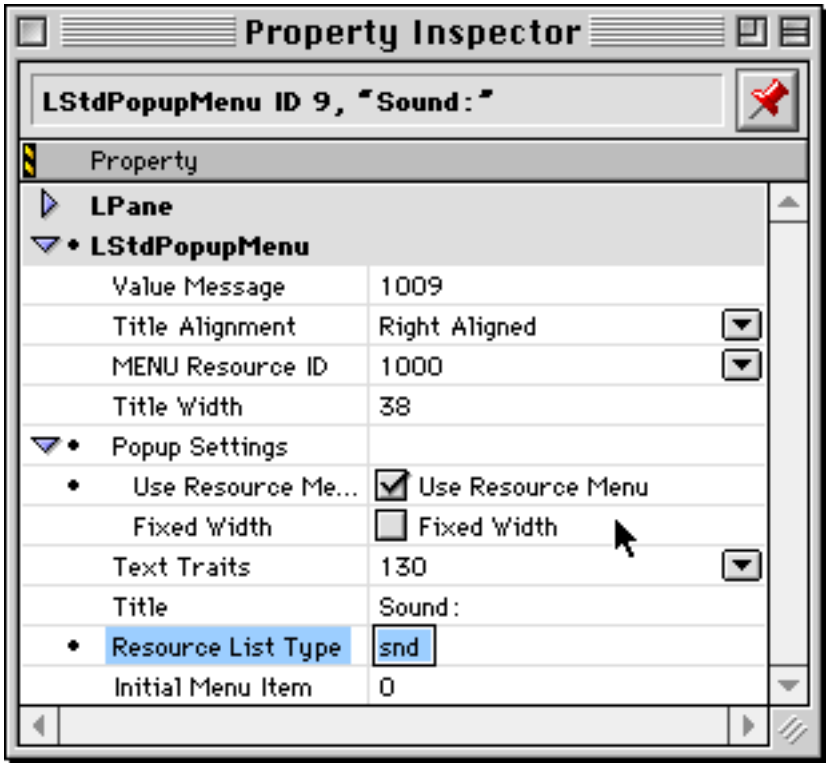
To complete the interface, you

- Modify the LStdPopupMenu
- Create the CColorControl custom type

1. **Complete the sound popup menu**

If you have not already opened the PPob project file, double-click the `Controls.ppob` file in the IDE project window. Constructor launches and the Constructor project window appears. Double-click the LWindow view to see its contents, and then double-click the menu item control. The Property Window for the LStdPopupMenu object appears. Most of the data has been set for you.

Figure 8.14 LStdPopupMenu popup properties



This menu contains a list of sound resources. Check the *Resource List Type* box, and enter the name of the type of resource you want to ap-

pear in the menu. In this case, it is "snd "—with a trailing space. This is the name of the resource type for sounds.

With that information set, PowerPlant will build the menu for you. You must still provide a MENU resource, but the MENU resource should be empty.

Close the Property Inspector window and save your changes.

2. Build the CColorControl object.

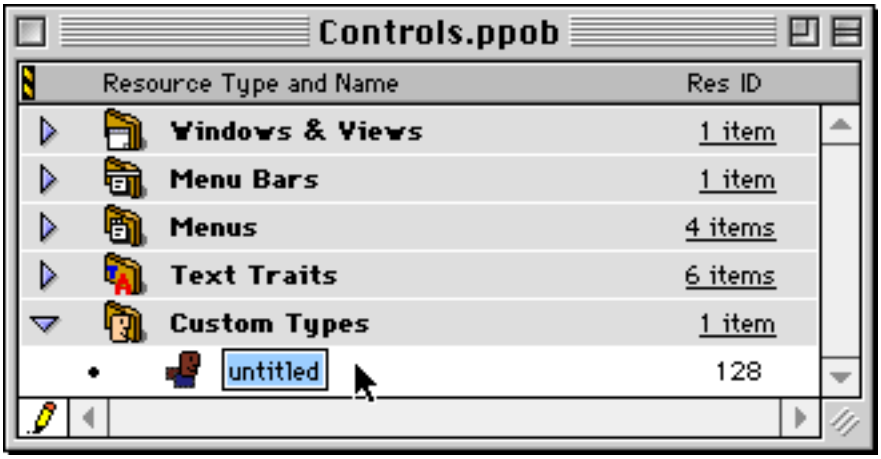
The CColorControl object is a custom pane type. It is a control, but it requires additional data. If you want to be able to set that data in Constructor in the Property Inspector window, you must create a CTYP resource for this custom pane.

You must create the CTYP resource, specify class information, specify the additional data items that appear in the Property Inspector window for this custom pane, add the object to the window, and set the object's characteristics.

a. Create a custom type.

While in the Constructor project window, select the Custom Pane Types heading, then choose **New Resource** (command-K) from the **Edit** menu. Constructor builds a new, untitled custom pane type resource (CTYP), as shown in Figure 8.15.

Figure 8.15 Creating a custom type

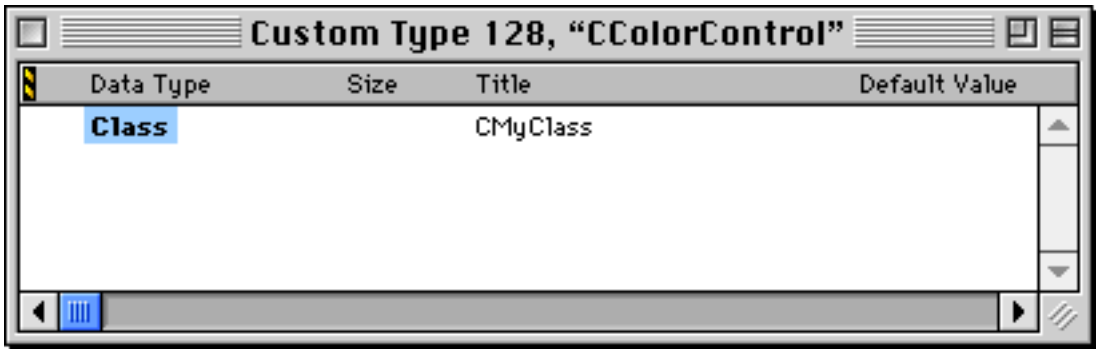


With the new resource selected as shown, set the resource name to CColorControl. The ID should remain 128.

b. Edit the class information.

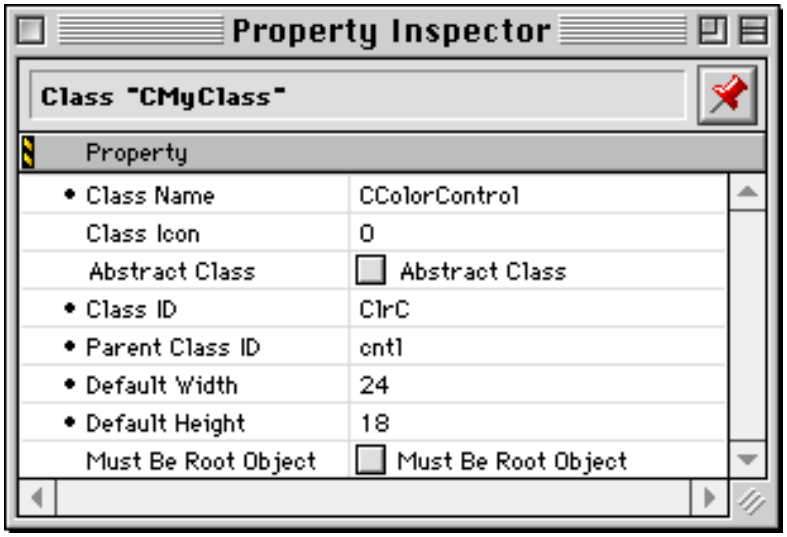
Double click the CColorControl resource in the project window. The CTYP editor window shown in Figure 8.16 appears.

Figure 8.16 The custom type view



Now, double-click the Class to set the new class's ancestor and other data. When you do, the window shown in Figure 8.17 appears.

Figure 8.17 The Class editor window



Set the data to match the illustration. Set the name, class ID, parent class ID, default height and width.

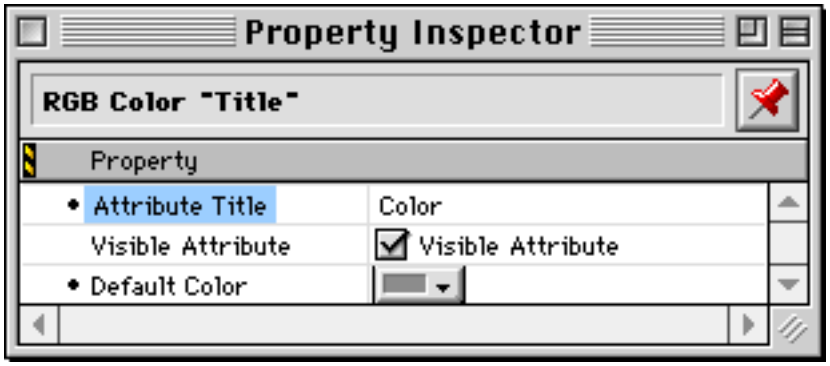
Close the window after you have made the changes.

c. Add data to the custom type.

This control stores a color. To allow yourself to set a color in this custom class's Property Inspector window, you add the necessary data item to the CTYP resource.

With the CTYP editor window active, choose **New RGB Color Item** from the **Custom Type** menu. Then double-click the new item to set its properties.

Figure 8.18 Setting a new RGB Color properties



Set the title to Color. The title is the label that will appear next to this item in the Property Inspector window for the custom pane. You can also pick a default color if you wish.

When you have set the data, close the CTYP editor window.

d. Add the object to the window.

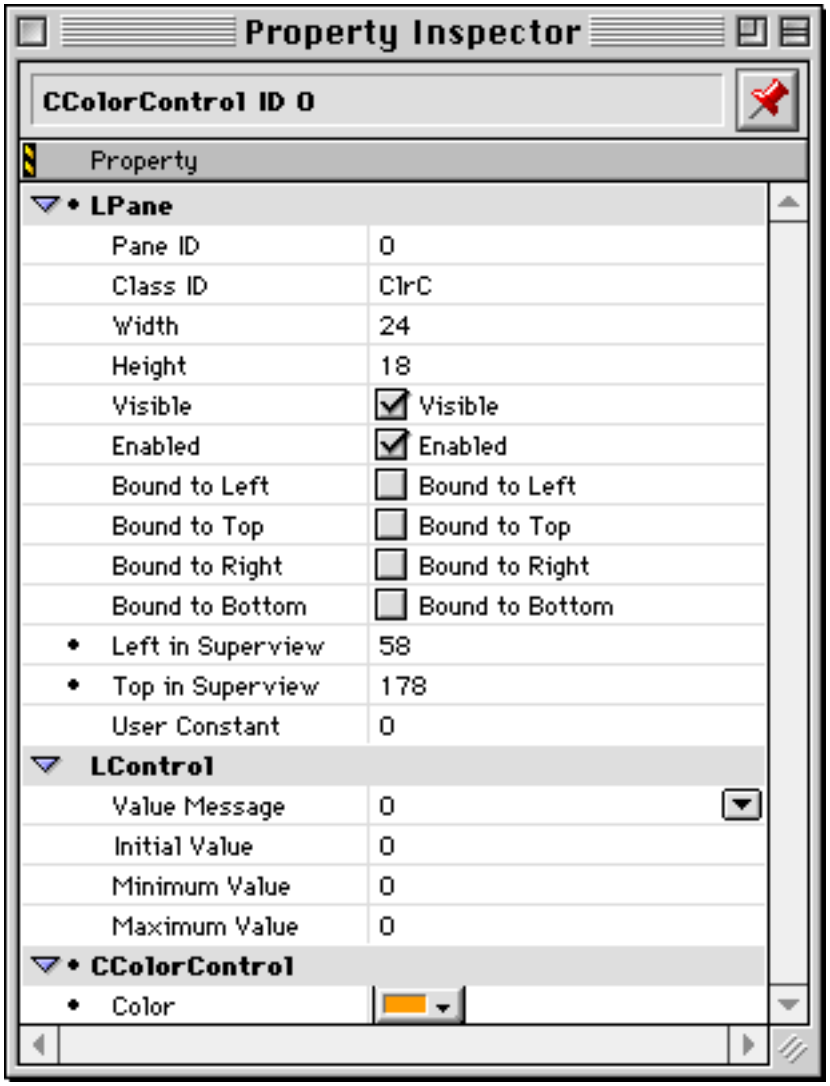
Double-click the PPob resource to open the layout editor. Make sure you also have the Catalog Window open as well, and showing the control classes. The CColorControl pane type should appear in the Catalog window.

Drag a CColorControl object into the layout.

e. Set the object characteristics.

Double-click the new CColorControl object in the layout window. The Property Inspector window should appear, as shown in Figure 8.19.

Figure 8.19 CColorControl Property Inspector



There is an item for the new data type you entered, the RGB Color. Set the characteristics of the pane to match the values in the illustration. Pay particular attention to the class ID and value message. You can set an initial color by clicking the color box.

Congratulations! You have just created a custom Constructor type. When you create a CTYP resource, you are really creating a template that Constructor can use for any number of objects of that par-

ticular class. You can copy and paste CTYP resources from project to project. So you can use your custom Constructor types in any project.

You must still write the code to implement this custom object as well as the other controls. That's what you'll do in the next steps.

CColorControl

In this section you implement the functions for CColorControl. The header for the CColorControl class is complete. It sets the class ID and declares the various functions. In the following steps you implement these functions:

- `CColorControl(LStream*)`
- `SetColor()`
- `DrawSelf()`
- `FindHotSpot()`
- `PointInHotSpot()`
- `HotSpotAction()`
- `HotSpotResult()`

The remaining constructors and destructor have been provided for you. There is a class creator function as well. It simply calls the stream constructor.

This is a long exercise, so don't forget to save your work often.

3. **Implement the stream constructor**

`CColorControl(LStream*)` `CColorControl.cp`

When you created the custom type, you added some extra data—an RGBColor. You must read that data out of the stream and initialize the related data member. The header for this class names the data member as `mColor`. We discuss streams in "What Is a Stream" on page 446.

The existing code calls the LControl stream constructor.

```
CColorControl::CColorControl( LStream *inStream )
    : LControl( inStream )
```

```
{
  // Initialize the color from the stream.
  inStream->ReadData( &mColor, sizeof(RGBColor) );
}
```

4. Write the SetColor() accessor.

`SetColor()` `CColorControl.cp`

This function receives the color as an input parameter. It should do two things. It should set the `mColor` data member, and update the control. You should refresh the control whenever the color changes to ensure that it reflects the current state of the object.

By the way, the `GetColor()` accessor is provided for you.

```
CColorControl::SetColor( const RGBColor &inColor )
{
  mColor = inColor;

  // Refresh control to reflect color change.
  Refresh();
}
```

5. Draw the control.

`DrawSelf()` `CColorControl.cp`

Every pane has a `DrawSelf()` function. Controls are no exception. This control draws a black frame, leaves a one-pixel white space, and then fills the rest of the control with the control's color. Of course, you should preserve the color state and set it to known values before drawing.

```
CColorControl::DrawSelf()
{
  // Save and normalize the color/pen states.
  StColorPenState savePenState;
  StColorPenState::Normalize();

  // Calculate the frame rect.
  Rect theFrame;
```

```
  CalcLocalFrameRect( theFrame );

  // Frame the control.
  ::FrameRect( &theFrame );

  // Draw a white area inner frame.
  RGBColor theWhiteColor = {0xffff,0xffff,0xffff};
  ::RGBForeColor( &theWhiteColor );
  ::InsetRect( &theFrame, 1, 1 );
  ::FrameRect( &theFrame );

  // Fill in rest of control with the color.
  ::RGBForeColor( &mColor );
  ::InsetRect( &theFrame, 1, 1 );
  ::PaintRect( &theFrame );
}
```

6. Identify the hot spot.

`FindHotSpot()` `CColorControl.cp`

PowerPlant uses this function to determine which hot spot, if any, contains the specified point. The CColorControl object has only one hot spot, the colored area of the control. If the point is inside the hot spot, return the value 1. Otherwise, return the value zero. The colored area is inset two pixels from the frame of the control. The function receives the point in question.

```
CColorControl::FindHotSpot( Point inPoint )
{
  Int16 theHotSpot = 0;

  // Calculate the frame rect.
  Rect theFrame;
  CalcLocalFrameRect( theFrame );

  // Inset to get the colored interior region.
  ::InsetRect( &theFrame, 2, 2 );

  // Check if the point is in our hot spot.
  if ( ::PtInRect( inPoint, &theFrame ) )
```

```
    theHotSpot = 1;

  return theHotSpot;
}
```

7. Determine if a point is in the hot spot.

`PointInHotSpot()` `CColorControl.cp`

PowerPlant uses this function to determine if a point is inside a particular hot spot. The function receives the point in question, and the number of the hot spot.

The CColorControl object has one hot spot, the colored area of the control. The colored area is inset two pixels from the frame of the control. You should return true if the hot spot is number 1, and the point is inside the control's hot spot.

```
CColorControl::PointInHotSpot( Point inPoint,
                               Int16 inHotSpot )
{
  Boolean theResult = false;

  // Calculate the frame rect.
  Rect theFrame;
  CalcLocalFrameRect( theFrame );

  // Inset to get the colored interior region.
  ::InsetRect( &theFrame, 2, 2 );

  // Check if the point is in our hot spot.
  if ( inHotSpot == 1 && ::PtInRect(inPoint, &theFrame ) )
    theResult = true;

  return theResult;
}
```

8. Act while the button is down in the hot spot.

`HotSpotAction()` `CColorControl.cp`

PowerPlant calls this routine repeatedly while the mouse button is down inside a hot spot. The CColorControl object draws a highlight while the button is in the control, and no highlight while the button is outside of the control. A more complex control could perform a more complex action, such as scrolling a view.

This function receives three parameters: the hot spot, and two Boolean values. If these values are the same, there has been no change. If they are different, then the mouse has moved either into or out of the hot spot and you should act accordingly.

Remember, CColorControl only has one hot spot. If you get any hot spot value besides 1, you should do nothing.

Because you are drawing in the control, you should call `FocusDraw()` before drawing to ensure that the drawing environment is set up properly.

```
CColorControl::HotSpotAction(Int16 inHotSpot,
                      Boolean inCurrInside,
                      Boolean inPrevInside )
{
  if ( inHotSpot == 1
        && inCurrInside != inPrevInside
        && FocusDraw() ) {

    // Calculate the frame rect.
    Rect theFrame;
    CalcLocalFrameRect( theFrame );

    // Inset it to account for the frame.
    ::InsetRect( &theFrame, 1, 1 );

    // If we're inside, draw the hilight black,
    RGBColor theColor;
    if ( inCurrInside ) {
      theColor.red = theColor.green = theColor.blue = 0x0000;
    } else { // erase it with white.
      theColor.red = theColor.green = theColor.blue = 0xffff;
```

```
    }
    ::RGBForeColor( &theColor );

    // Draw the hilight.
    ::FrameRect( &theFrame );
  }
}
```

9. Act when the button is released in the hot spot.

`HotSpotResult()` `CColorControl.cp`

PowerPlant calls this routine when the mouse button is released inside a hot spot. The function receives a single parameter, the number of the hot spot involved.

If the hot spot is number 1, the CColorControl object should unhighlight the control. You should then display the standard color selection dialog. Set the new color and broadcast a message to all listeners that the color has changed.

```
CColorControl::HotSpotResult( Int16 inHotSpot )
{
  // Only act if we're in the hot spot.
  if ( inHotSpot == 1 ) {
    // Undo hilighting.
    HotSpotAction( inHotSpot, false, true );

    // Choose a new color.
    Point thePoint = {-1,-1};
    RGBColor theNewColor;
    if ( ::GetColor( thePoint, "\pChoose a color:",
                     &mColor, &theNewColor ) ) {

      // Set the new color.
      SetColor( theNewColor );

      // Broadcast the message.
      BroadcastValueMessage();
```

```
  }
}
```

Save your work and close the file.

The `BroadcastValueMessage()` call is perhaps the single most important line of code you wrote in this section. This is the call that tells all the listeners that something happened.

In the remaining steps you hook listeners to controls to make everything work.

The Controls Application

In the design of this application, there are three levels of objects with varying degrees of responsibilities. There is the application, the window, and the controls. You could make either the window or the application listen to messages.

In this case we use the application as the principal listener. CControlsApp inherits from both LApplication and LListener.

Listing 8.3 CControlsApp class declaration

```
class CControlsApp : public LApplication, public LListener {
public:
            CControlsApp();
  virtual   ~CControlsApp();

  virtual void ListenToMessage( MessageT inMessage,
                                void *ioParam );

protected:
  LWindow* MakeControlsWindow();

private:
  LWindow* mWindow;
};
```

We discuss application objects in detail in the next chapter. Don't be alarmed about getting ahead of ourselves. Your work is really with this particular application's "listener" nature.

This application object overrides `ListenToMessage()` inherited from LListener, has a new function `MakeControlsWindow()`, and a new data member that points to the only window.

In the following steps you will:

- Register custom classes
- Link the application to its controls.
- Link the radio buttons to the check box.
- Make the application respond to the controls.
- Make the radio buttons respond to the check box.

10. Add include file for class registration

`Top of File` `CControlsApp.cp`

To register any PowerPlant or custom class, you need to include the header file for that class in your main source file.

```
// include header for LToggleButton
#include <LToggleButton.h>

// Custom Classes to be registered
#include "CColorControl.h"
#include "CRadioButton.h"
```

11. Register custom class.

`CControlsApp()` `CControlsApp.cp`

This application uses two custom classes, CColorControl and CRadioButton. In addition, this application uses LToggleButton. LToggleButton and the two custom classes must all be registered individually.

```
  RegisterClass_(LRadioGroup);

// Register additional PowerPlant classes.
  RegisterClass_(LToggleButton);
```

```
  // Register custom classes.
  RegisterClass_(CColorControl);
  RegisterClass_(CRadioButton);

mWindow = MakeControlsWindow();
```

The existing code also calls `MakeControlsWindow()` to build the window. This is where you begin the real work of adding functionality to the application.

12. Link the application to all controls.

`MakeControlsWindow()` `CControlsApp.cp`

The existing code calls the LWindow class creator function. After that, you link the application to all controls in the window.

Remember, Constructor creates a RidL resource listing all the controls in a window. The file `ControlsConstants.h` declares `rRidL_ControlsWindow` to match that resource ID number.

```
theWindow = LWindow::CreateWindow( rPPob_ControlsWindow, this );

// Link the application (the listener) with the
// controls in the window (the broadcasters).
UReanimator::LinkListenerToControls( this, theWindow,
                                     rRidL_ControlsWindow );
```

13. Link the radio buttons to the check box.

`MakeControlsWindow()` `CControlsApp.cp`

In the design of the Controls application, the check box enables or disables the radio buttons. To make that happen, the start code CRadioButton class inherits from both LStdRadioButton and LListener. To complete the links, you must make the CRadioButton objects listen to the check box object.

To do this, get the check box object using `FindPaneByID()`. Then find each radio button object in turn using `FindPaneByID()`. Tell the check box to add that object as a listener. The constants listed in the source code are declared in `ControlsConstants.h`.

```
  UReanimator::LinkListenerToControls(this, theWindow,
                                      rRidL_ControlsWindow );

  // Get the check box.
  LStdCheckBox *theCheckBox;
  theCheckBox = dynamic_cast<LStdCheckBox *>
                (theWindow->FindPaneByID( kStdCheckbox ));

  // Get radios make them listen to check box.
  CRadioButton *theRadio;
  theRadio = dynamic_cast<CRadioButton *>
             (theWindow->FindPaneByID( kStdRadio1 ));
  theCheckBox->AddListener( theRadio );

  theRadio = dynamic_cast<CRadioButton *>
             (theWindow->FindPaneByID( kStdRadio2 ));
  theCheckBox->AddListener( theRadio );

  theRadio = dynamic_cast<CRadioButton *>
             (theWindow->FindPaneByID( kStdRadio3 ));
  theCheckBox->AddListener( theRadio );

theWindow->Show();
```

WARNING! There is no error control at all here. This code assumes that each of the calls to `FindPaneByID()` returns a valid pointer and the `dynamic_cast` succeeds. This is not wise. In robust code you would check the validity of the returned pointer. We'll discuss PowerPlant's debugging features in the next chapter.

The existing code then shows the window.

14. Make the application respond to controls.

`ListenToMessage()` `CControlsApp.cp`

As you know, each listener has a single `ListenToMessage()` function in which it responds to messages. The CControlsApp object is no exception. The application object should respond in two ways.

First, for every message received, display the message in the message caption. To accomplish this task, get the message caption by telling the window to `FindPaneByID()`. The constant for the pane ID is `kMessagePane`. After you have the pointer to the caption, set the caption's value, and refresh the caption so the window redraws. For an LCaption the value is the descriptor.

Second, if the message comes from either the sound button or the sound popup menu, play a sound. To accomplish this task, identify the appropriate message. If that message is received, get a pointer to the popup menu object. Read the value of the popup menu, and play the corresponding sound.

The solution code does some of the work for you. It converts the message into a string. The code for playing the sound already exists as well. You do the work in between. Make sure you use the same local variable names as the solution code where necessary.

```
#pragma unused( ioParam )

// set message in message pane
LCaption* theCaption = dynamic_cast<LCaption*>
                        (mWindow-> FindPaneByID(kMessagePane));
ThrowIfNil_( theCaption );
theCaption->SetValue(inMessage);
theCaption->Refresh();

switch ( inMessage ) {
// identify sound messages
  case msg_PlaySoundButton:
  case msg_SoundPopup:
  {
    // Get the popup menu.
    LStdPopupMenu *thePopup;
    thePopup = dynamic_cast<LStdPopupMenu *>
```

```
            (mWindow->FindPaneByID( kStdPopupMenu ));

    // Get the name of the sound to play.
    Str255 theSoundName;
    ::GetMenuItemText( thePopup->GetMacMenuH(),
                       thePopup->GetValue(), theSoundName );
```

The existing code then gets and plays the sound resource.

Save your work and close the file.

15. Make the radio buttons respond to the check box.

`ListenToMessage()` `CRadioButton.cp`

The radio buttons listen to any message from the check box. In fact, the check box sends one message. The `ControlConstants.h` file declares `msg_EnableDisable` to match the "able" message specified in Constructor.

The radio button receives this message whether the check box is turning on or off. When the radio button receives this message, check the state of the radio button. If it is enabled, disable it. If it is disabled, enable it.

```
switch ( inMessage ) {  // identify and respond to message
  case msg_EnableDisable:
    if ( IsEnabled() ) {
      Disable();
    } else {
      Enable();
    }
    break;
```

Save your work and close the file.

Great news! You have completely implemented a custom control, linked an application to a set of controls, and linked some controls to another control. Let's watch how it all works.

16. Build and run the application.

Make the project and run it. When you do, a window should appear containing all the views. See Figure 8.13 on page 248. Play with the controls in the window and watch what happens.

Click on each control. Watch the message that's displayed near the bottom of the window. Feel free to check in Constructor to see how the messages match the value message. If you're careful, you're going to notice an interesting fact.

The message received when you click on a radio button or a text button is not the value message! The default behavior for these controls broadcasts the `msg_ControlClicked` value, which is 203. The `PowerPlant_Messages.h` file declares this constant.

Click the check box and observe the message. The number is the numerical representation of the "able" message. Notice how the radio buttons respond to the check box. This is your code at work.

Choose a sound from the sound popup menu. The sound should play. The application hears the message and responds. Click in the sound button. The sound should play again.

Click the LToggleButton. Observe how the picture in the button changes. PowerPlant steps through the series of graphics provided for the button. Observe similar behavior when you click the LCicnButton. Here you have two states, pushed and not pushed.

Finally, don't forget the CColorControl. After all the work you did writing the code for that class, you might as well get some enjoyment out of it. Play with the control. Click and hold the mouse button down while moving the mouse in and out of the control. Highlighting should turn off and on appropriately. Click the control, and the standard color picker dialog should appear. Pick a color, and the button should change to reflect your choice.

Notice that the application receives the message from the CColorControl object. Stop and think about what you've got here. You now have a ready-made PowerPlant class that you can drop into any PowerPlant project any time you need a control to select a color. Because it broadcasts a message, *any* dependent object that listens to

the message can change its color in response. Cool! This could be really useful.

If you would like to experiment further, here's a suggestion. Notice that if you use the sound popup menu and choose the current sound (that is, you make no change in the menu), the sound does not play. That's because the default behavior for the menu does not send a message unless the popup menu changes. If you wanted to use this in a real application, you might want to modify that behavior so that a sound plays when the user chooses the current item in the menu. You'll have to subclass LStdPopupMenu to make this happen. If you're feeling adventurous, don't let that stop you. Go for it, and have a good time.

Intermission

Well done! You have done some serious PowerPlant programming. You have not only learned all about controls, you have also finished the Basic Building Blocks section of the manual. It's time to take a breather and look at where you've been, where you are, and where you're going to go in the rest of the manual.

In the first section of the manual you learned the fundamentals of application framework design. You got a broad panorama of framework architecture. You learned about design patterns, and how PowerPlant implements those patterns in a carefully-crafted Macintosh application framework. You also learned that an application framework is first and foremost a mechanism for managing the visual interface in an application.

In the Basic Building Blocks section that you have just finished, you learned all about the fundamental building blocks you use to build a visual interface in PowerPlant. You have mastered panes, views, and controls. You know how they relate to each other, how to create them, how they are typically used, and what they're good for.

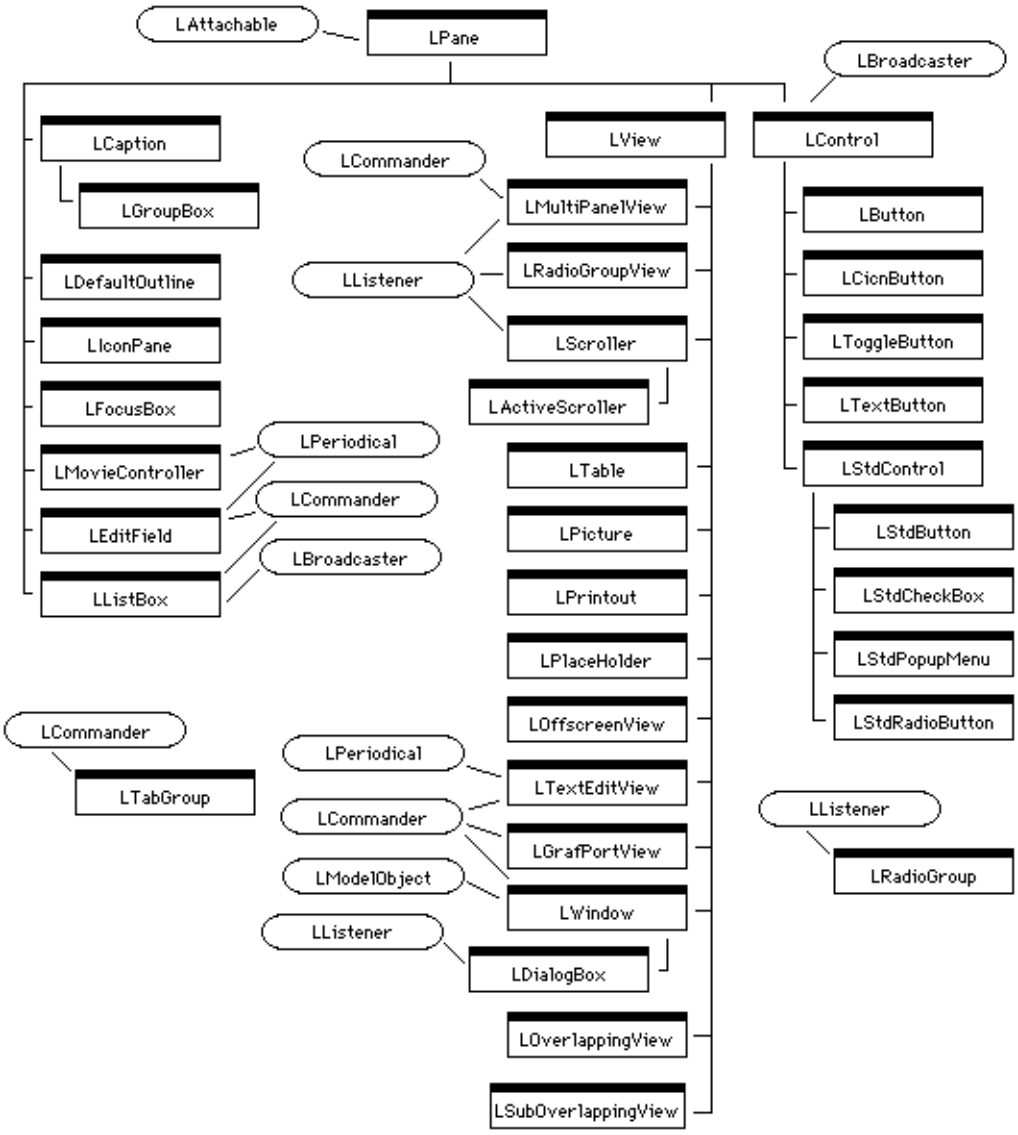
That's a lot of knowledge! Take a look at Figure 8.20 to get the big picture. We haven't talked about all of these classes in detail, but we have discussed most of them.

In the process we have spent a fair amount of time deep in the details of code. Every now and then you got a glimpse of the higher principles being implemented, like the messaging system. And that has helped you keep the big picture in mind.

In the next section of the manual we're going to talk about how a PowerPlant application really works. We'll start with the command hierarchy. Then we're going to return to the visual hierarchy again. Only this time, rather than talking about the fundamental building blocks, we're going to put those blocks together in windows and dialogs.

Then we'll discuss other application tasks, like file I/O and printing. Along the way you're going to see and learn a lot more about PowerPlant.

Figure 8.20 The LPane hierarchy



9

Applications and Events

In this chapter we begin the process of building an application in PowerPlant. In the Basic Building Blocks chapters we used a class-centered perspective to learn about the pane classes. We talked all about LPane, LView, LControl, and their descendants.

From now on we're going to use a task-based perspective. We won't be able to avoid talking about classes, but for the most part you're going to see these discussions centered around how to use a class to accomplish a particular programming task.

In this chapter we discuss:

- The Application Object—the class hierarchy, the individual classes, and deriving your own application class.
- Initializing an Application—initializing your own application, including memory management and debugging in PowerPlant.
- Event Handling and Dispatch—how it works, and why you won't need to change it very often.

As usual, you will close out the chapter with a code exercise.

The Application Object

The application class is the basic foundation of a PowerPlant application. In this section we discuss the fundamental information you need to know to understand and use an application object, including:

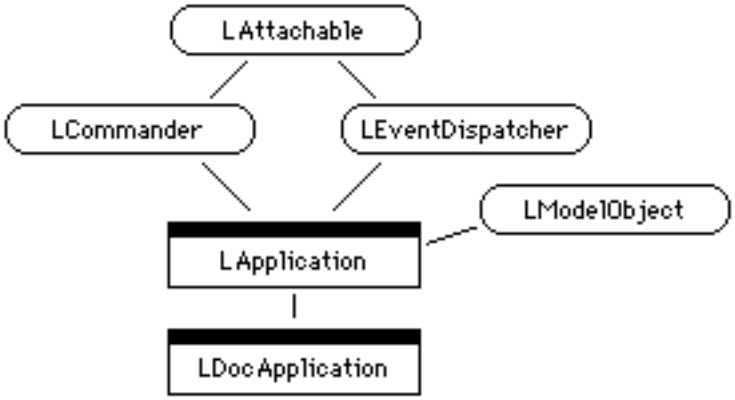
- Application Class Hierarchy
- Application State

- Deriving an Application

Application Class Hierarchy

Figure 9.1 illustrates the class hierarchy. There are only two application classes in PowerPlant, LApplication and LDocApplication.

Figure 9.1 Application class hierarchy



The LDocApplication class is effectively identical to LApplication, except that it provides member functions to support documents and printing. We will discuss documents in Chapter 13, "File I/O" and printing in Chapter 14, "Printing" later in this manual.

LApplication is the basis of our discussion in this chapter. We are going to concentrate on setting up an application, and on the event handling features of an application.

Notice that LApplication inherits from several mix-in classes. Each of these is the source for part of the nature of an application:

LEventDispatcher—an application receives, identifies, and dispatches events. (See "Event Handling and Dispatch" on page 290.)

- LModelObject—an application receives and handles Apple events, and is scriptable. (See "PowerPlant and Apple Events" on page 291.)

- LCommander—an application handles commands from the user (or other parts of the application). (See Chapter 10, "Commanders and Menus.")
- LAttachable—an application may have attachments. Attachments add behavior to an application. (See Chapter 15, "Periodicals and Attachments.")

In addition to the behavior it inherits or overrides from its various base classes, an application has a few functions that it implements for itself. Table 9.1 lists each function and its purpose.

Table 9.1 LApplication functions

Function	Purpose
`GetState()`	return what the application is doing
`SetSleepTime()`	set amount of time to surrender to background processes
`Run()`	process events until quitting
`ProcessNextEvent()`	retrieve and dispatch events
`ShowAboutBox()`	display the application's About box
Initialize()	perform setup work such as create/modify menus
`StartUp()`	respond to an Open Application Apple event
`SendAEQuit()`	send a Quit Application Apple event to self
`DoQuit()`	attempt to quit the application

Application State

An application's state reflects what it is doing. Listing 9.1 contains the three possible states the application may be in.

Listing 9.1 EProgramState enumeration

```
enum EProgramState {
  programState_StartingUp,
  programState_ProcessingEvents,
  programState_Quitting
};
```

PowerPlant uses these values internally, and typically you won't be concerned with or modify an application's state.

If you want to quit the application, you can do so by setting the application's `mState` data member to `programState_Quitting`. At the risk of repeating ourselves, PowerPlant's default behavior takes care of this for you. You shouldn't have to do this on your own. In fact, doing so may bypass features like checking to save changed documents before allowing the application to quit.

Deriving an Application

Typically you derive your own application class from either LApplication or LDocApplication. Most of an application's standard behavior, as implemented by PowerPlant default member functions, will suit you just fine.

You will certainly override some LApplication member functions, including:

- `ShowAboutBox()`—because you're going to want a killer About Box.
- `StartUp()`—to respond to an open application Apple event in your own way—for example, to display a splash screen.
- `Initialize()`—last chance to initialize Application before processing events.

In addition, you'll want to override some member functions inherited from LCommander including

- `ObeyCommand()`
- `FindCommandStatus()`

We'll talk about these last three functions in the next chapter. In the rest of this chapter we'll talk about setting up an application and LApplication's event handling features.

Initializing an Application

Every C++ program needs a `main()` function. A PowerPlant application is no exception. PowerPlant itself does not have a `main()`, but the `CHyperApp.cp` file does. When you use PowerPlant stationery to create a new project, this file is automatically included in the project. Listing 9.2 contains the code for this `main()` function. We'll use it as an example throughout this chapter.

When you create a new project from stationery, you should open the `CHyperApp.cp` file and save it with a new name in your project's folder. This will automatically update the CodeWarrior project file at the same time. You can then make necessary changes to the file.

Alternatively, you can create your own version of `main()` that suits your own needs.

Listing 9.2 A sample main()

```
void main(void)
{
  // Set Debugging options
  SetDebugThrow_(debugAction_Alert);
  SetDebugSignal_(debugAction_Alert);

  // Initialize Memory Manager - Parameter is
  // number of Master Pointer blocks to allocate
  InitializeHeap(3);

  // Initialize standard Toolbox managers
  UQDGlobals::InitializeToolbox(&qd);

  // Install a GrowZone function to catch
  // low memory situations.
```

```
  new LGrowZone(20000);

  // replace this with your App type
  CHyperApp theApp;
  theApp.Run();
}
```

Initializing and launching an application requires that these tasks be performed as necessary:

- Set Debugging Options
- Initialize the Heap
- Initialize the Toolbox
- Setup Memory Management
- Check the Environment
- Register PowerPlant Classes
- Run the Application

Let's look at each of these tasks. We're going to take the opportunity to explore some utilitarian features of PowerPlant in great detail, particularly debugging and memory management.

Set Debugging Options

PowerPlant has powerful debugging macros that you can use when developing software. You can read about these features in two files: `UDebugging.h` and `UException.h`. You should also consult *PowerPlant Revealed*.

You can use PowerPlant's debugging features without using any other part of PowerPlant. Simply include the header files and corresponding source files in your project.

The PowerPlant debugging-macro strategy is straightforward. You may "throw" an error or raise a signal. The "throw" occurs if you use the `Throw_` macro. This is similar the C++ keyword `throw`, and in fact, eventually performs a C++ `throw`. `Throw_` performs an extra notification of the error condition through a dialog or low-

level debugger. We will use the macro name `Throw_` to avoid confusion. In addition to `Throw_` there are four signal macros that we'll refer to generically as `Signal_`.

WARNING! PowerPlant debugging support requires the presence of a debugger like MacsBug (or other low-level debugger such as TMON or Jasik) or the CodeWarrior IDE. If you do not have such a debugger present, you will crash.

An error indicates something is wrong. A `Signal_` indicates something unusual has happened or just an informative message. If you ignore an error, something bad happens. If you ignore a `Signal_`, nothing bad happens.

In general, you should `Throw_` when an error occurs. An error may be defined as a situation which, if not handled, can cause significant problems (like crashing the computer.) You should raise a `Signal_` for a condition which does not threaten the integrity of the application or the stability of the computer, but which is unusual.

For example, you might want to `Throw_` an error if you encounter a nil handle. You might want to raise a `Signal_` if you try to unlock an already unlocked handle.

To activate these macro capabilities, you `#define` two terms: `Debug_Throw` and `Debug_Signal`. The effects of defining or not defining these terms are listed in Table 9.2.

Table 9.2 Effect of debugging options

Term	Macro	action
`Debug_Throw` defined	`Throw_`	`Throw_` invoked
`Debug_Throw` not defined	`Throw_`	`throw` C++ exception
`Debug_Signal` defined	a signal	macro invoked
`Debug_Signal` not defined	a signal	nothing

In either case—`Throw_` or one of the four `Signal_` macros—PowerPlant defines four possible actions to take, as shown in Listing 9.3.

Listing 9.3 Debug action enumeration

```
typedef enum {
  debugAction_Nothing =0,
  debugAction_Alert = 1,
  debugAction_LowLevelDebugger = 2,
  debugAction_SourceDebugger = 3
} EDebugAction;
```

The alert action displays a dialog containing the exception code, as well as the source code file name and line number that generated the `Throw_` or `Signal_`. The low-level debugger action displays a string—in MacsBug for example—identifying the routine and offset into the routine, and the exception code. The source-level debugger stops with the current statement arrow pointing to the line containing the `Throw_` or `Signal_`.

WARNING! If you use the low-level debug action and don't have a low-level debugger installed, you will crash when you `Throw_` or `Signal_`. If you use the source-level debug action and don't have a source-level debugger running, you may crash, or you may break into a low-level debugger if one is available. We recommend you have a low-level debugger like MacsBug installed at all times.

TIP: It is best, for compatability purposes, not to use the source-level debug action. This avoids Mixed Mode Manager switches when debugging on PowerPC Mac OS computers and keeps your debugging information in tact.

PowerPlant maintains two global variables, `gDebugThrow` and `gDebugSignal` that specify which of the four possible actions to take on either a `Throw_` or a `Signal_`. By default, `gDebugThrow` and `gDebugSignal` are set to `debugAction_Nothing`. You can set

their values at any point in the program if you want to use different options in different sections of code. Usually, you set their values at the beginning of your main program.

For example, this code from `main()`...

```
SetDebugThrow_(debugAction_Alert);
SetDebugSignal_(debugAction_Alert);
```

...uses macros to set the global variable to a debug action.

Having defined `Debug_Throw` and `Debug_Signal` and having set the values for `gDebugThrow` and `gDebugSignal`, you can use the macros defined in `UDebugging.h`. Table 9.3 lists their usage.

Table 9.3 PowerPlant debugging macros

Macro	Usage
`Throw_()`	perform debug action for a `Throw_`
`SignalPStr_()`	raise a signal, pass a Pascal string
`SignalCStr_()`	raise a signal, pass a C string
`SignalIf_()`	raise a signal if condition is true
`SignalIfNot_()`	raise a signal if condition is false
Assert_()	same as `SignalIfNot_()`

Using these macros is pretty straightforward, with one exception. Remember that if `Debug_Signal` is not defined, then any signal macro does nothing. This includes the popular `Assert_` macro. When your code is compiled, every occurence of a signal macro generates no code.

As a result, you should be very careful that the test inside the signal macro have no side effects. A test with side effects can lead to very subtle bugs creeping into your code when you turn debugging off. Take a look at this sample code.

Listing 9.4 A bad Assert_ test

```
void main(void)
{
  int number = 5;
  Assert_(--number < 10);
  cout << "number = " << number << '\n';
}
```

The test in the `Assert_` macro has the side effect of decrementing the value in `number`. When `Debug_Signal` is defined, the macro test executes, and `number` has a value of 4. When `Debug_Signal` is not defined, the macro generates no code, and `number` has a value of 5. The code does not run the same when debugging is off.

`UException.h` defines several more macros that are especially useful for Mac OS programming. The macros for throwing exceptions defined in `UException.h` all eventually invoke the underlying `Throw_` macro.

Table 9.4 More PowerPlant debugging macros

Macro	Usage
`ThrowIfOSErr_()`	test for OSErr, `Throw_` if non-zero result
`ThrowOSErr_()`	`Throw_` an error (use when you already know there's an error)
`ThrowIfNil_()`	`Throw_` if the parameter is nil
`ThrowIfNULL_()`	`Throw_` if the parameter is nil
`ThrowIfResError_()`	calls `ResError()`, `Throw_` if non-zero result
`ThrowIfMemError_()`	calls `MemError()`, `Throw_` if non-zero result
`ThrowIfResFail_()`	check a Resource Manager handle; if nil, `Throw_` an error

Macro	Usage
`ThrowIfMemFail_()`	check a Memory Manager pointer or handle; if nil, `Throw_` an error
`ThrowIf_()`	`Throw_` if condition is true
`ThrowIfNot_()`	`Throw_` if condition is false
`FailOSErr_()`	same as `ThrowIfOSErr_()`
`FailNIL_()`	same as `ThrowIfNil_()`

You will find these macros used throughout the PowerPlant source code. You can use them in your own code as well. When you want to turn off all the debugging code, simply comment out your definition of `Debug_Throw` and/or `Debug_Signal`. If you turn off `Debug_Throw`, all of your `Throw_` macro calls will automatically call the standard C++ `throw()`.

TIP: PowerPlant has several stack-based classes for memory management. The advantage of stack-based objects is that the destructor is automatically called, even when there is an exception thrown. See "Stack-based memory classes" on page 285.

A `Throw_` occurs inside a `Try_` block. The `Try_` macro is the PowerPlant equivalent of the C++ `try` keyword. They are identical.

Of course you must have something to handle a `Throw_`. The PowerPlant macro is `Catch_()`. It maps directly to the C++ catch mechanism but will only catch a throw of type `ExceptionCode`. For example:

```
Catch_(iErr)
```

and

```
catch ( Exceptioncode iErr )
```

are equivalent. If you need a "catch all," use the C++ `catch()`.

If you do not have a `catch()` handler of some type, your program will terminate. The `LApplication::Run()` function has a universal catch handler, so a PowerPlant application isn't likely to terminate. However, that handler simply displays a message.

TIP: Grab a good C++ book and look over the C++ exception handling mechanisms such as `new_handler`, `unexpected()`, `terminate`, etc. Remember, even though you are using the PowerPlant framework, you are still using in the C++ Language. All of the benefits and features of the C++ language are available to you...feel free to explore and use them!

None of the PowerPlant debugging features prevents you from using the standard C++ `try`, `throw()`, `catch()` exception handling mechanism. The PowerPlant macros map to this mechanism. However, the PowePlant macros offer an additional layer of information and debugging capability.

TIP: Although not related to PowerPlant, CodeWarrior also includes DebugNew—a utility for debugging memory allocation. Examine the file `DebugNew.cp` for additional debugging features regarding the `new` operator.

Initialize the Heap

The second task you must perform when launching an application is to initialize the application's heap. The sample `main()` function calls `InitializeHeap()`. This function is defined in `UMemoryMgr.cp`. It is not a member of any class.

Call this function at the beginning of your program (before initializing the Toolbox) to expand the heap zone to its maximum size and allocate a specified number of master pointer blocks. If you want to perform unusual tasks, such as modifying the size of the stack, you can replace `InitializeHeap()` or call your own function in addition, as appropriate.

Initialize the Toolbox

After initializing the application's memory space, you must set up the Mac OS Toolbox for your application's use. PowerPlant provides the UQDGlobals class to handle this for you. Our sample `main()` function calls `UQDGlobals::InitializeToolbox(&qd)`.

This function initializes all the common Toolbox managers, as shown in Listing 9.5.

Listing 9.5 UQDGlobals::InitializeToolbox() snippet

```
::InitGraf((Ptr) &sQDGlobals->thePort);
::InitFonts();
::InitWindows();
::InitMenus();
::TEInit();
::InitDialogs(nil);
```

If you have additional managers to initialize, you must do so at this phase of the startup process. You may override the UQDGlobals class, but a more typical solution would be to simply put the necessary code either in your own function or directly in `main()`.

TIP: If your application uses QuickTime, initialize the QuickTime Manager with `UQuickTime::Initialize()`.

Setup Memory Management

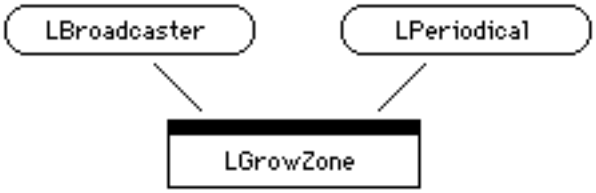
A good application framework provides significant assistance for managing memory. PowerPlant certainly qualifies.

PowerPlant establishes an emergency memory reserve and uses a `GrowZone()` function to release it. PowerPlant also provides a series of memory-related classes for helping you with typical memory-related housekeeping.

The LGrowZone class

LGrowZone encapsulates the PowerPlant memory management strategy. It has two parts: a memory reserve, and a strategy for asking other objects to release memory. Figure 9.2 illustrates the inheritance hierarchy for LGrowZone.

Figure 9.2 LGrowZone hierarchy



Notice that LGrowZone is one of PowerPlant's free-standing classes. You can use LGrowZone by including nothing more than it, LPeriodical, LBroadcaster, and LListener in your projects.

In your application you create a single instance of LGrowZone. When you call the LGrowZone constructor, you specify the size of the memory reserve, as our `main()` function does with this code:

```
new LGrowZone(20000);
```

Notice that LGrowZone inherits from LBroadcaster. When faced with a memory shortage, LGrowZone does two things. It asks all listeners to release memory, and it releases the memory reserve.

Objects that are able to free memory when needed should be listeners. You attach them to the LGrowZone object so that the listeners are notified when memory is low, using code like this:

```
LGrowZone::GetGrowZone()->AddListener(myObject);
```

When memory runs short, the object receives a `ListenToMessage()` call with a `msg_GrowZone` message and a pointer to the number of bytes still needed. The object can then respond accordingly. The object tells LGrowZone the number of bytes freed. If it

cannot release any memory, it must supply a zero so LGrowZone knows that memory was not released.

LGrowZone also inherits from LPeriodical. We discuss LPeriodical in greater detail in Chapter 15, "Periodicals and Attachments." In a nutshell, a periodical object is called from the main event loop either on every pass through the loop, or when null events are received.

In this case, LGrowZone's constructor sets up the object so that its `SpendTime()` function is called each time through the main event loop. In that function, if the reserve has been used up, LGrowZone tries to re-establish the reserve. If it cannot, it warns the user.

Consult *PowerPlant Revealed* for more details on LGrowZone.

Stack-based memory classes

PowerPlant gives you a set of simple classes to help you allocate and deallocate memory safely. These classes are all declared in `UMemoryMgr.h`.

In these utility classes, the constructor performs some action and the destructor undoes the action. The advantage of stack-based objects is that the destructor is automatically called, even when there is an exception thrown.

Table 9.5 Effect of stack-based memory classes

Class	Constructor	Destructor
StHandleLocker	locks a given handle	restores the handle's locked/unlocked state
StHandleBlock	allocates a relocatable block	deallocates the block
StClearHandle Block	allocates a relocatable block full of zeros	deallocates the block

Class	Constructor	Destructor
StTempHandle	allocates a relocatable block in temporary memory	deallocates the block
StPointerBlock	allocates a non-relocatable block	deallocates the block
StClearPointerBlock	allocates a clear non-relocatable block	deallocates the block
StResource	gets the handle for the specified resource	releases the resource handle
StHandleState	gets the handle state	restores the handle state

Note that the StHandleLocker class does **not** move the handle high in the heap.

The StHandleBlock class may use temporary memory if the application's heap is full. StClearHandleBlock does not.

There are two other block-related functions you may wish to use, `BlocksAreEqual()` and `BlockCompare()`. See *PowerPlant Revealed* for details.

Finally, UMemoryMgr.h also declares the StValueChanger template class. The constructor saves the original value and changes to the specified new value. The destructor restores the original value. This is a useful class for preserving and restoring state information.

Other possible memory strategies

PowerPlant does not use a memory pre-flighting strategy where memory requests are tested against available memory before an attempt is made to allocate the memory.

If you would like to implement memory pre-flighting, or use a memory management strategy with a finer resolution than simply releasing the memory reserve in one fell swoop, you may certainly

do so. You may derive your own memory management class from LGrowZone, or create your own.

Check the Environment

Although not obvious, our sample `main()` function does a simple assessment of the operating environment. Your application may need to do more. Our `main()` creates an instance of the application object.

```
CHyperApp theApp;
```

In the process, its constructor and the default LApplication constructor are called. In the `LApplication::LApplication()` constructor, PowerPlant looks for the version of QuickDraw in the environment (among other tasks). See Listing 9.6.

Listing 9.6 Lapplication::LApplication() snippet

```
    // Check for Color QuickDraw
  Int32 qdVersion = gestaltOriginalQD;
  ::Gestalt(gestaltQuickdrawVersion, &qdVersion);
  UEnvironment::SetFeature(env_SupportsColor,
                (qdVersion > gestaltOriginalQD));
```

PowerPlant uses the UEnvironment class to track several features, as shown in Listing 9.7.

Listing 9.7 PowerPlant environment tracking

```
enum {
  env_SupportsColor        = 0x00000001,
  env_HasDragManager       = 0x00000002,
  env_HasThreadsManager    = 0x00000004,
  env_HasThreadManager     = 0x00000004,
  env_HasAOCE              = 0x00000008,  // obsolete
  env_HasStdMail           = 0x00000010,  // obsolete
  env_HasStdCatalog        = 0x00000020,  // obsolete
  env_HasDigiSign          = 0x00000040,  // obsolete
```

```
  env_HasQuickTime        = 0x00000100,
  env_HasAppearance       = 0x00001000,
  env_HasAppearanceCompat = 0x00002000,
  env_HasAaron            = 0x00004000,
  env_HasAppearance101    = 0x00008000   // AM 1.0.1 installed?
};
```

You can inquire if a feature is available by calling `UEnvironment::HasFeature()`, a static member function.

The application constructor is a good place for application-level initialization and environment testing. For example, The `LApplication()` constructor builds the menu bar. In your own constructor you may wish to do additional testing for other features of the environment, modify the application's sleep time, and so forth. You could also perform these tasks in a separate initialization function immediately after creating an application object if you wish.

See also the *PowerPlant Revealed* HTML reference for more information on UEnvironment.

Register PowerPlant Classes

Your application object's constructor typically performs one more critical function—it registers the necessary PowerPlant classes. PowerPlant relies heavily on stream-based creator functions in its visual (UI) classes. The PPob resource encapsulates the information necessary to build you UI elements from scratch using this stream-based creator function technique.

When creating a PPob-based object, PowerPlant must know which creator function to call for which class. It does this by maintaining a table that associates a unique class ID with that class's stream-based constructor. You *must* have an entry in this table for the class creator function before instantiating a PPob-based object.

To register an individual class, you use the `RegisterClass_()` macro. You provide your class name, the macro does the work of creating a creator function and adding it to the class table. Your class must have a unique class ID in the class definition.

WARNING! If you do not register each and every class that you directly utilize in your PPob resource(s), your application will not work properly. If you have the `Signal_` debugging features turned on, you should get a "Unregistered ClassID" signal raised in `URe-animator.cp` when you try to instantiate a class without first registering that class.

Failing to register a class is perhaps the single, most common cause of problems encountered by new PowerPlant programmers.

Here's an example call to `RegisterClass_()` that registers the LButton class.

```
RegisterClass_(LButton);
```

It is *not* necessary to register PowerPlant classes that are not explicitly used in your PPob resource(s).

For example, if you have a class called CMyPane that inherits from LPane.You use CMyPane in your PPob but never use LPane (directly). You do of course need to register CMyPane, but you do *not* need to register LPane as you never directly utilize LPane in your PPob. In this case, you still need to include `LPane.cp` in your project since CMyPane inherits from it.

WARNING! Previously, to register an individual class, you had to call `URegistrar::RegisterClass()` and provide the class ID and the creator function. This method is obsolete and should not be used. The obsolete method is currently supported for compatibility with existing classes, but will not be supported in the future. You should update your code accordingly.

Finally, *don't forget that you must register any PPob-based class that you derive*.

Run the Application

After you create your application object, perform any additional initialization, and register each and every PPob-based class you use, it's time to start the main event loop running. Our sample `main()` function does the following to accomplish this task:

```
theApp.Run();
```

Just call the application object's `Run()` function, and you're on your way. The `Run()` function makes the menu bar and calls the `LApplication::Initialize()` function to perform additional setup such as modify the menus in the application. We'll discuss this task in the next chapter.

Event Handling and Dispatch

The application's `Run()` function calls `ProcessNextEvent()` repeatedly. `ProcessNextEvent()` does the real work. It:

- Adjusts the cursor.
- Gets the next event.
- Sends the event to any attachments for pre-processing.
- Dispatches the event if it needs further processing.
- Distributes idle time if the event is a null event.
- Calls periodical items in the queue that receive time on every pass through the event loop.
- Updates menus if necessary.

Because LApplication inherits from LEventDispatcher, it can dispatch events. It calls the inherited `DispatchEvent()` function. This function parses the event and calls the appropriate handler.

Each handler performs whatever additional parsing (if any) is necessary. The handler might identify the most recently clicked pane, or retrieve the current target object in the command hierarchy. You can study the LEventDispatcher code to see the details.

The handler dispatches the event to the appropriate object. A click goes to a pane. A command or keystroke goes to a commander. The pane or commander is responsible for handling the event.

If a commander does not handle a command itself, it passes the command back up the command chain until someone does handle it. The ultimate supercommander is your application object. It is responsible for any command not handled by objects below it in the command hierarchy. We'll discuss this process in more detail in Chapter 10, "Commanders and Menus."

Before we do, notice that this section on events and dispatching does not include any instructions for typical ways in which you override or derive classes to modify the default event-dispatch behavior of PowerPlant. While you are certainly free to do so, it is unlikely that you will ever need to modify event handling and dispatch. This behavior is a gift from PowerPlant. Enjoy it.

However, there is one tricky detail that occasionally trips up new PowerPlant programmers—Apple events.

PowerPlant and Apple Events

PowerPlant relies on Apple events for some of its basic functionality. The process of launching an application is a good example. When you launch an application from the Finder, after the application launches it receives one of three Apple events from the Finder—open application, open documents, or print documents.

Assuming that you aren't opening or printing documents, the application receives the open application Apple event. In response to that event, the application calls the application's `StartUp()` function.

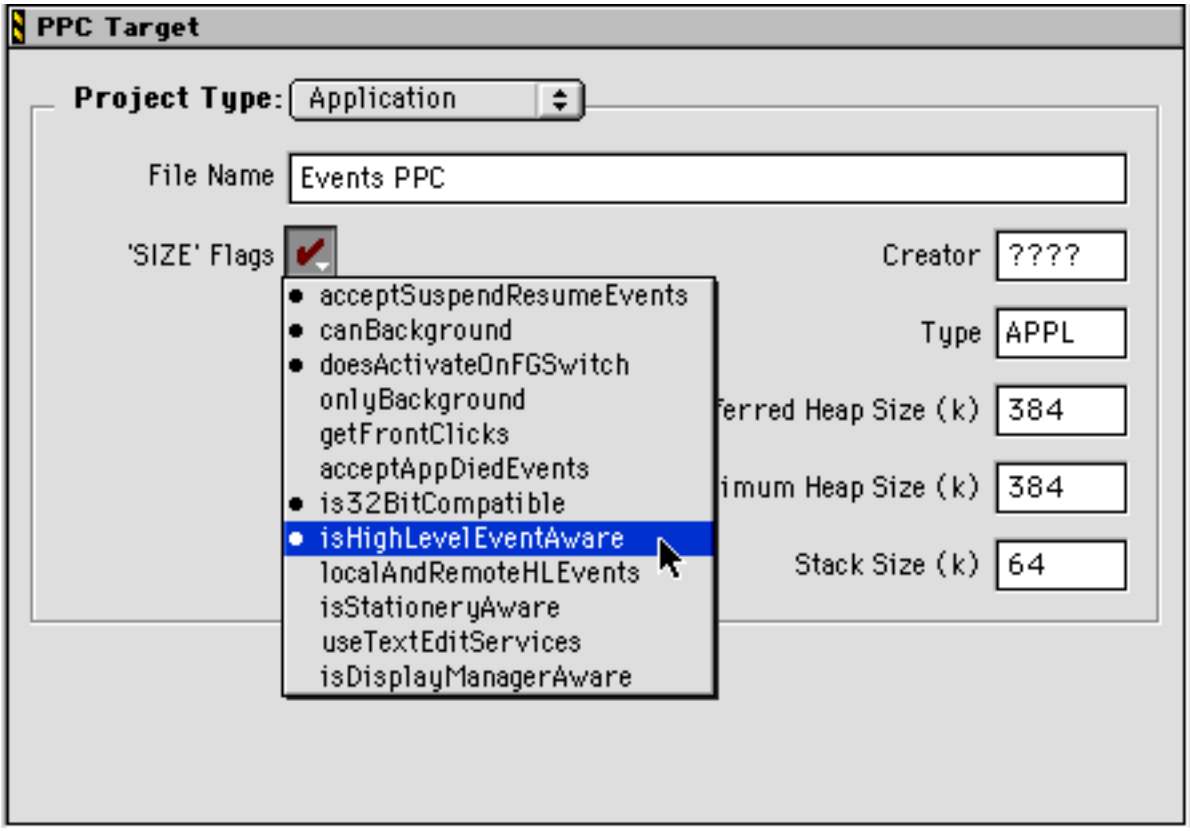
In this function you can perform some setup work for the application. For example, you might want to have a default window open on launch if the user isn't opening documents. There are any number of tasks you might perform in the `StartUp()` function.

However, for this to work your application must be aware of Apple events. To ensure that it is, go to the **68K Processor** or **PPC Processor** target settings panel (depending on your build target), and ex-

amine the SIZE flags. Make sure the "isHighLevelEventAware" flag is checked(Figure 9.3). If it is not, your application will not receive Apple events, and `StartUp()` won't be called.

You must also have the proper 'aedt' resources in your application. If you use PowerPlant project stationery, the file `PP AppleEvents.rsrc` is contains these resources and is included for you.

Figure 9.3 isHighLevelEventAware flag in target preferences



Summary

In this chapter you learned how to initialize a PowerPlant application, and about the PowerPlant utilities designed to help you in that task.

PowerPlant has a powerful set of macro-based debugging features for throwing exceptions or raising signals. These features are implemented in an independent section of PowerPlant so that you can use the debugging features without using the rest of PowerPlant.

PowerPlant uses an emergency reserve memory management strategy in association with a well-designed LGrowZone object. The LGrowZone object broadcasts a need for memory. Objects you create that can release memory can listen to the LGrowZone object and respond to the plea for memory donations. You also learned about the many stack-based utility classes that can assist you in robust memory management.

You learned how to register PowerPlant pane classes, and about the importance of registering your own classes as well. Finally, you read about the event dispatch mechanism in PowerPlant.

Let's see how it all works in some real code.

Code Exercise

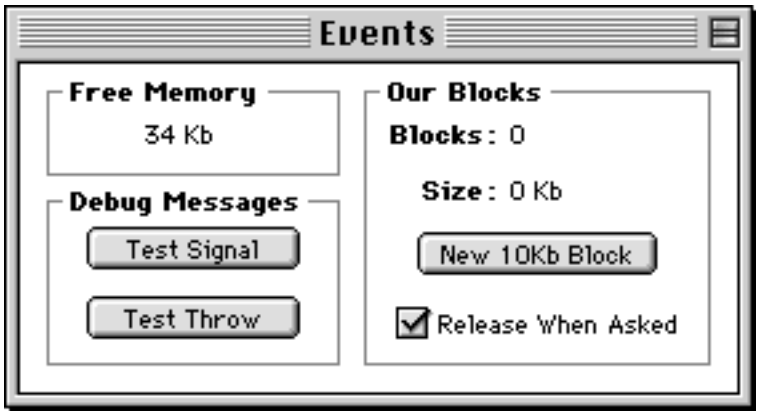
The application you work on in this exercise is named "Events." Because event dispatch is completely implemented in PowerPlant, this chapter emphasizes debugging and memory management. We also look at how the project sets up the development environment. These details are important when you want to turn debugging on, register new PowerPlant classes, and so forth.

As usual, we'll look at the application interface first, and then write code to implement the application.

The Interface

The Events application is a memory eater. Figure 9.4 shows you what the application looks like.

Figure 9.4 The Events application



Examine the various panes in Constructor. In this code exercise the PPob resource is complete. You won't add or modify any panes.

There are three LGroupBox objects, of which we take no further note.

There are four LCaption objects. Two of them are simple labels. Two of them report information about the application's use of memory. The "Blocks" caption reports the number of blocks you have created. The "Size" caption reports how much memory the application has eaten.

The "Free Memory" caption is a custom caption pane. It reports available memory. You'll register this class when you write the application. The code for the class is provided for you.

There are three buttons. The **Test Signal** button sends a signal. The **Test Throw** button throws an exception. The **New 10Kb Block** button creates a memory block, if memory is available. The application keeps track of the allocated blocks.

The Release When Asked button controls the application's behavior when it runs out of memory. At that time, the LGrowZone object will ask its listeners for memory. The application listens for the message and responds.

Like the code you wrote in Chapter 8, the application listens to the controls in its window and responds appropriately. You'll write some of that code in this exercise.

Setting Up an Application

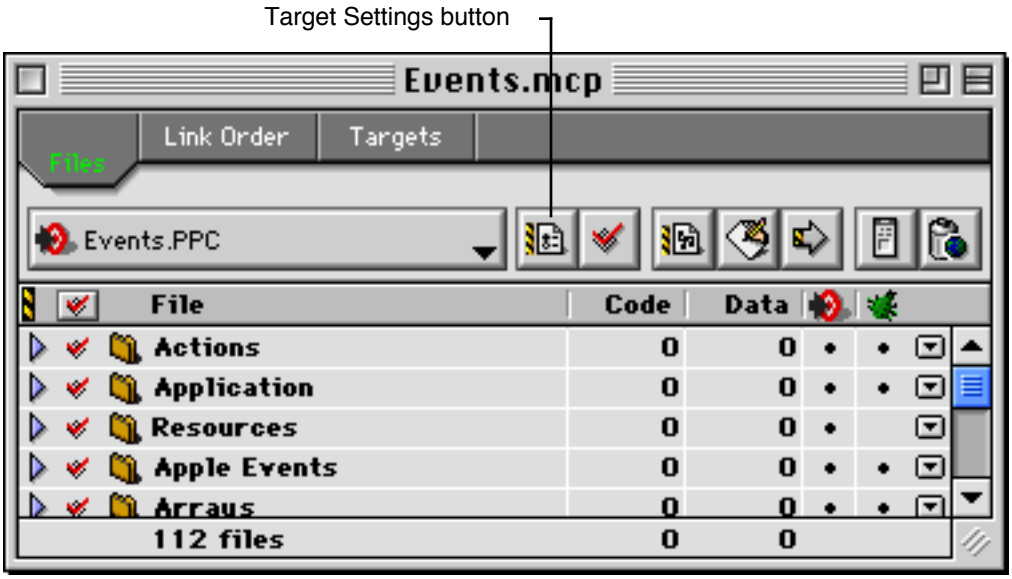
In this section you set up the development environment, set up debugging, and implement the Events application.

1. **Examine project prefix.**

`C/C++ settings dialog` `Events.mcp`

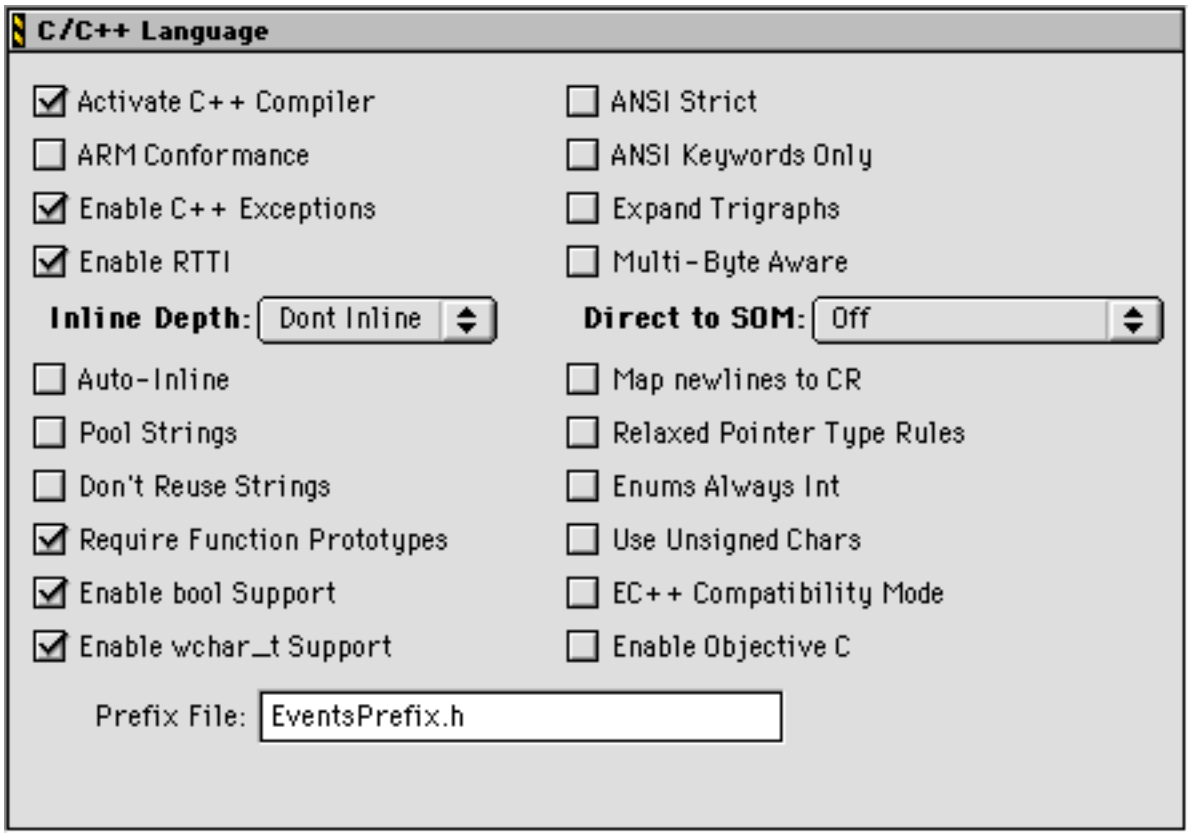
Click the **Target Settings** button on the `Events.mcp` project window (Figure 9.5). Then choose the C/C++ Language panel, as shown in Figure 9.6.

Figure 9.5 Target Settings button



TIP: The toolbar on the IDE Project Window and the global toolbar are completely customizable. See the IDE User Guide for more information on how to customize CodeWarrior.

Figure 9.6 Events language settings



Notice that the prefix file is `EventsPrefix.h`. In the PowerPlant stationery, the prefix file is `PP_DebugHeaders68K` or `PP_DebugHeadersPPC`. The `PP_DebugHeaders.cp` file used to build those precompiled headers contains the following code:

```
// Define all debugging symbols
#define Debug_Throw
#define Debug_Signal

// include the header files for the standard PowerPlant classes
#include <PP_ClassHeaders.cp>
```

One way to turn on debugging is to use `PP_DebugHeaders` for the correct build target (68K or PowerPC). To turn debugging off,

change the project prefix to `PP_MacHeaders68K` or `PP_MacHeadersPPC`.

You'll accomplish the same goal in `EventsPrefix.h` in just a bit.

2. **Set up the environment**

`no function` `EventsPrefix.h`

You accomplish one primary in this step. You turn on debugging. The existing code includes the correct version of `PP_MacHeaders` depending on the build target.

```
#include <PP_MacHeaders.h>

// Define debugging symbols.
#define Debug_Throw
#define Debug_Signal

// Include the PowerPlant prefix file.
#include <PP_Prefix.h>
```

The `PP_Prefix.h` file defines some additional terms for the universal headers, and includes additional PowerPlant header files.

Save your work and close the file.

3. **Examine the application class**

`class declaration` `CEventsApp.h`

This particular application class inherits from LApplication and LListener. This allows the application to listen to the controls in the window. It also allows the application to listen to the LGrowZone object. Listing 9.8 contains the complete class declaration.

Listing 9.8 CEventsApp class declaration

```
class CEventsApp : public LApplication, public LListener {
public:
                CEventsApp();
  virtual       ~CEventsApp();

  virtual void ListenToMessage( MessageT inMessage,
```

```
                                    void *ioParam );

protected:
  LWindow*      MakeEventsWindow();

private:
  LWindow*      mWindow;
  LArray        mBlockList;
};
```

This class overrides the `ListenToMessage()` function inherited from LListener. It declares a new function, `MakeEventsWindow()` to create the window. It has two new data members, `mWindow` and `mBlockList`. The former points to the application's only window. The latter is a list of allocated blocks of memory.

When you are through examining the declaration, close the file.

4. Build the application object

`CEventsApp()` `CEventsApp.cp`

The project's `main()` function is provided for you in this file. It initializes the heap, initializes the Toolbox, creates the LGrowZone object, and creates an application object. Of course, the application constructor is called at that moment.

In this step you write the application constructor. There are five tasks you must accomplish. They are:

a. Set the action to occur on `Throw_` and `Signal_`.

Use the `SetDebugThrow_` and `SetDebugSignal_` macros. Set the response to `debugAction_Alert` so an alert is displayed in response to either a throw or signal.

b. Register all PowerPlant classes.

Call `RegisterAllPPClasses()`.

c. Register any custom classes.

The only custom class is the CFreeMemoryCaption class.

d. Make a window.

Call the application's `MakeEventsWindow()` function. It returns the `LWindow*`. You can store it in `mWindow`. And, now that you have debugging features available, use a macro to check the LWindow pointer and ensure it isn't nil.

e. Link the application to the LGrowZone object.

Get the LGrowZone object and call its `AddListener()` function. You want to link the application object to the LGrowZone object.

The solution code is listed here for reference. This function is empty in the start code. You write the whole thing here.

```
// Setup the throw and signal actions.
SetDebugThrow_( debugAction_Alert );
SetDebugSignal_( debugAction_Alert );

// Register PowerPlant class creator functions.
RegisterAllPPClasses();

// Register custom classes.
RegisterClass_( CFreeMemoryCaption );

// Create the single application window.
mWindow = MakeEventsWindow();
ThrowIfNil_( mWindow );

// Listen to messages from the grow zone.
LGrowZone::GetGrowZone()->AddListener( this );
```

5. Make the Events window.

`MakeEventsWindow()` `CEventsApp.cp`

The PPob that describes this window has been provided for you. In this step you accomplish two tasks.

a. Create the window.

Call `LWindow::CreateWindow()`, the class creator function. The resource ID constant is `rPPob_EventsWindow`. The window's supercommander is the application object.

This call returns a pointer to an LWindow object. Use a macro to ensure that the pointer is not nil. You can use `Assert_`.

b. Link the application to the controls in the window.

Call `UReanimator::LinkListenerToControls()`. The ID constant for the RidL resource is `rRidL_EventsWindow`.

The solution code for both tasks is listed here.

```
// Create the window.
LWindow* theWindow;
theWindow = LWindow::CreateWindow( rPPob_EventsWindow, this );
ThrowIfNil_( theWindow );

// Link the application (the listener) with the
// controls in the window (the broadcasters).
UReanimator::LinkListenerToControls( this, theWindow,
                                     rRidL_EventsWindow );

theWindow->Show();
```

The existing code shows the window and returns the LWindow pointer to the caller—the application constructor in this case.

6. Respond to messages.

`ListenToMessage()` `CEventsApp.cp`

The application listens to the controls in the window. It also listens to the LGrowZone object. The application's `ListenToMessage()` function may receive four messages:

- `msg_GrowZone`
- `msg_NewBlock`
- `msg_TestSignal`
- `msg_TestThrow`

In this step you write some of the code to respond to the `msg_GrowZone` message, and all the code for testing the signal and throw mechanisms. The code for creating a memory block is provided for you.

a. Release memory if appropriate.

When you receive a `msg_GrowZone` message, send yourself a signal. Although you wouldn't do this in a real application, this is instructional here. When the LGrowZone object asks for memory, you'll hear about it.

```
case msg_GrowZone:
{
  // We're asking for memory, let user know
  SignalPStr_("\pLGrowZone asking for memory" );

  Int32 theBytesFreed = 0;
```

WARNING! Raising a signal here may cause a crash! This call causes PowerPlant to display a dialog. That dialog must be loaded into memory. Under just the wrong low-memory conditions, (and you'll be creating a low memory condition) there isn't enough room and a crash will result. You can omit this line of code without affecting the application. You just won't receive a notice when LGrowZone sends this message.

You should examine the remaining code in this `case`, it is very instructive. The code first determines the state of the check box. If memory release is allowed, the application walks through the list of memory blocks and releases them.

The code creates an iterator, starts with the first handle in the list, and operates on each handle. If the block is not a protected block, it removes the handle from the list and disposes of the handle. Notice that the iterator can modify the list while iterating! This causes no problems in PowerPlant. (See "Arrays" on page 547.)

When complete, the application updates the two captions that reflect the number of blocks and the amount of memory in those

blocks. Notice that the code uses an `LStr255` object. This is a descendant of LString. Among other features, the LString class provides a series of operator overloads for working with strings. The code here takes advantage of the LString features to automatically convert a 32-bit number into a string, and uses the += operator to append strings. (See "LString" on page 558.)

b. Send a signal.

When the application receives `msg_TestSignal`, send a signal.

```
case msg_TestSignal:
{
  // Raise a test signal.
  SignalPStr_( "\pSignal test" );
}
break;
```

c. Throw an exception.

When the application receives `msg_TestThrow`, throw an exception. You can use PowerPlant macros for the entire process. Create a `Try_` block. In the block, use `Throw_` to throw an error. Use the value -1. In the `Catch_` block, you don't have to do anything. This is just a test.

```
case msg_TestThrow:
{
  Try_ {
    // Throw an error.
    Throw_( -1 );
  } Catch_( inErr ) {
    // Catch it here.
  } EndCatch_
}
break;
```

That's it. Save your work and close the file.

The code to handle `msg_NewBlock` has been provided for you. Study it as another example of list management and string manipulation in PowerPlant.

7. **Build and run the application.**

Make the project and run it. When you do, a window should appear containing all the views. See Figure 9.4 on page 294.

Notice the amount of free memory, the number of allocated blocks, and the space required for those blocks. The Release When Asked check box should be on.

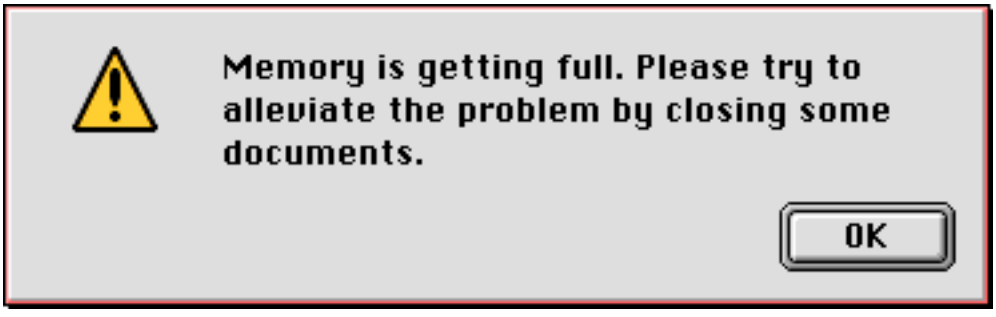
Click the **New 10Kb Block** button. Watch how the free memory, size, and number of blocks all change. Click the button repeatedly until you run out of memory. When you do, a signal dialog should appear telling you that LGrowZone is asking for memory.

NOTE: This is where you might crash. If the signal dialog does not appear and you crash, return to Step 6a and remove the code that calls `SignalPString()`.

If the signal dialog appears, click OK. Then observe the amount of free memory, number of blocks, and size.

Now, turn off the Release When Asked check box. The application will no longer release memory when asked. Create new blocks until you run out of memory. Once again, you should see the signal dialog telling you that LGrowZone is asking for memory. Click OK. This time, memory is not released. You have run out of memory and you'll see a warning dialog, as shown in Figure 9.7.

Figure 9.7 The low-memory alert



This is the standard PowerPlant low-memory alert. It isn't 100% appropriate in this circumstance, because there are no documents to

close. In your own application, you can modify this alert to display a more accurate message.

TIP: The memory warning should never cause a crash, even in low memory conditions. These warning dialogs are preloaded and locked, so they are always available.

Don't forget to test the two Debug Messages buttons to observe the signal and throw dialogs.

If you want to explore, turn off signaling and see what happens when you run out of memory. Turn off all debugging and see what happens when you throw an exception. Use different PowerPlant macros to test values, send signals and throw exceptions. Use the standard C++ `try`, `throw`, and `catch` keywords rather than PowerPlant macros, and see if there's any difference.

Finally, if you look at the free-memory caption closely you may notice it flash every second or so. The code for this custom caption was provided for you. Feel free to explore that code. This caption inherits from LPeriodical and installs itself in a special queue. Every time there is an idle event, this caption's `SpendTime()` function is called. We'll discuss how this works in "Periodicals" on page 506. If you explore this process, think about how you might eliminate or reduce the flashing. There is one straightforward solution. You could store the previous value, and only update the caption when the value changes. Implement that solution, or your own solution, and observe the difference.

Once again, congratulations are in order. This chapter deals primarily with low-level coding details like memory management and debugging.While this isn't the most glamorous part of PowerPlant, these are critical skills in the real world of software development. Best of all, you have now explored how to use them most effectively in PowerPlant, and practiced those skills.

In the next chapter you start working on menus. After that, you'll implement windows and dialogs. Then you'll work with docu-

ments, files, and printing. Finally, you'll study periodicals and attachments. The exciting stuff is coming!

10

Commanders and Menus

Now that you have an application set up and ready to run, the next step is to make it responsive. As the user makes menu choices, your application should respond appropriately. That responsiveness comes from the LCommander class.

This chapter has two main sections:

- Introduction to Commands—all about LCommander and its features.
- Making and Managing Menus—setting up menus and responding to menu choices in a PowerPlant application.

There are several important classes in PowerPlant that inherit from LCommander. Let's start there.

Introduction to Commands

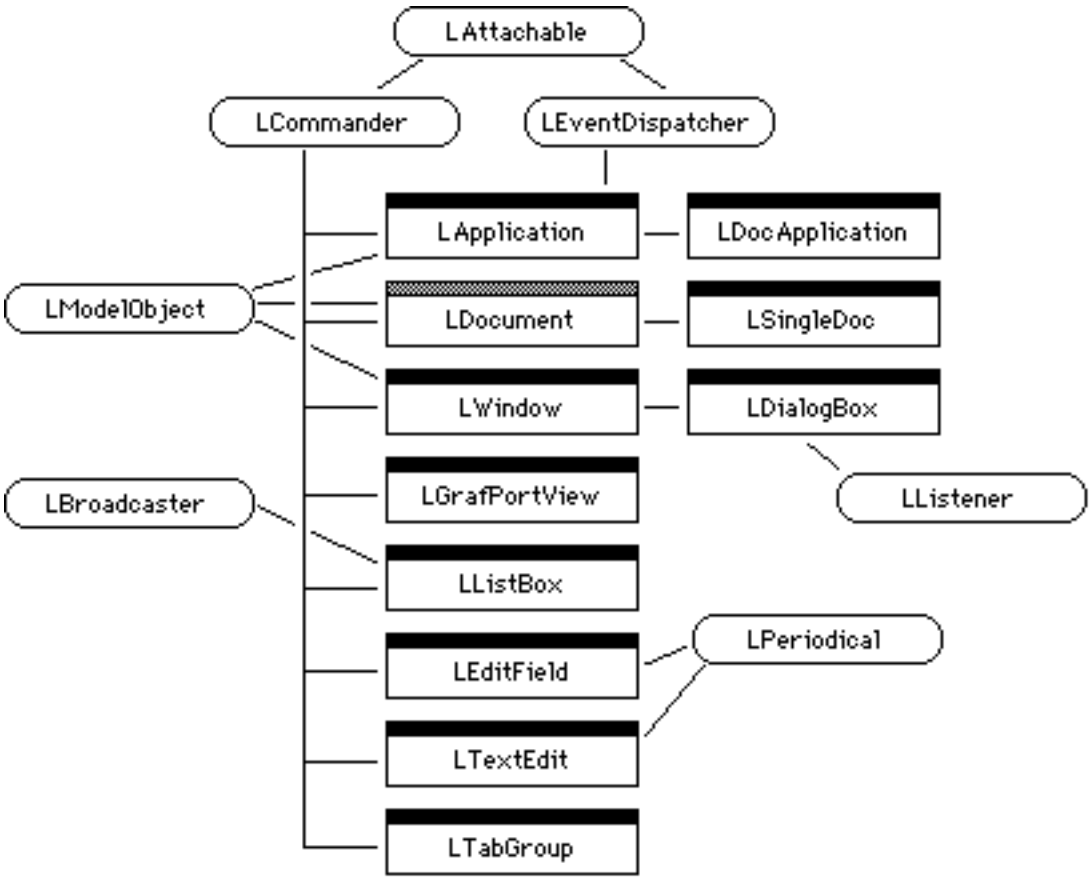
In most of our class hierarchy diagrams, we use LCommander as a mix-in class. This is appropriate, because classes in several different inheritance chains also inherit from LCommander.

Figure 10.1 shows what the class hierarchy looks like with LCommander as the principal base class. This diagram puts LCommander at the center of attention and illustrates all the classes that are also commanders.

Like the distinction between the LView class hierarchy and the application's visual hierarchy, there is a distinction between the LCommander class hierarchy (which simply illustrates inheritance) and the command hierarchy within a running application. We will

call the flow of commands within an application the “command chain.”

Figure 10.1 LCommander hierarchy



Remember that the subclasses in this diagram belong to various class hierarchies in PowerPlant, and many inherit from other classes besides LCommander, such as LPane or LView.

The LCommander class has functions devoted to:

- Command Chain—maintaining the command hierarchy.
- Target Handling—what commander receives a command.
- Duty Handling—which command chain is on duty.
- Command and Keystroke Handling—responding to menu commands and keystrokes.

Every commander—each object that derives directly or indirectly from LCommander—has these features.

Command Chain

In an application, the application object is the topmost commander. The LApplication constructor sets the application object as the top commander. The application object has no supercommander.

Every other commander has one supercommander. The LCommander class stores a pointer to an LCommander object—the supercommander—in the `mSuperCommander` data member. You use member functions to access this data.

Every commander may have an arbitrary number of subcommanders. The `mSubCommanders` data member is an LArray object. Table 10.1 contains the LCommander functions for command chain maintenance.

Table 10.1 Command chain maintenance functions

Function	Purpose
`GetTopCommander()`	return pointer to top commander
`SetSuperCommander()`	set the supercommander (replaces existing supercommander)
`AddSubCommander()`	add a subcommander to list
`RemoveSubCommander()`	remove a subcommander from list
`AllowSubRemoval()`	return whether to allow removal of subcommanders
`AttemptQuit()`	ask all subcommanders whether it is OK to quit
`AttemptQuitSelf()`	return whether it is OK to quit

Function	Purpose
`GetDefaultCommander()`	return pointer to default commander
`SetDefaultCommander()`	set default commander

`GetTopCommander()` is a static member function, so you can use `LCommander::GetTopCommander()` at any time to get a pointer to the application object.

The concept of the default commander deserves special attention. The default commander will be the supercommander for a commander created with a class creator function. PowerPlant uses the default commander when creating objects of various pane classes that are also derived from LCommander. These classes are LEditField, LListBox, LTextEditView, LWindow, LDialogBox, and LGrafPortView. Before creating one of these objects on the fly with a class creator function, you should ensure that the default commander is set appropriately.

In a subclass you may wish to override `AttemptQuitSelf()`. This function should handle any duties the commander must perform before quitting. LDocument is the only PowerPlant commander class that overrides this function. It does so to ensure that the user has an opportunity to save a changed document before quitting. We'll discuss saving documents in Chapter 13, "File I/O."

With that exception, the default functions for command chain maintenance are usually sufficient for most commanders. However, you may have noticed that there is no command dispatch mechanism. That's what the concept of the target object is all about.

Target Handling

At any moment there is one and only one target object. The target object is stored in a static LCommander data member—`sTarget`. Because it is a static class variable, there is only one instance of `sTarget`. When the application receives a command, it dispatches the command directly to the target commander, bypassing the com-

mand chain completely. Table 10.2 lists the target handling functions.

Table 10.2 Some LCommander target handling functions

Function	Purpose
`GetTarget()`	return pointer to current target
`SwitchTarget()`	change target
`AllowTargetSwitch()`	return whether to allow change in target
`IsTarget()`	return if the specified commander is the target
`BeTarget()`	called when commander becomes the target
`DontBeTarget()`	called when commander stops being target

`GetTarget()` and `SwitchTarget()` are both static member functions. You can call them from outside the class by using the class specifier—`LCommander::GetTarget()`, and `LCommander::SwitchTarget()`.

You use `SwitchTarget()` to make a new object the target object. For example, the LEditField pane is also a commander. When it receives a click, it calls `SwitchTarget(this)` to make itself the target.

In some cases, you may not want to allow the target object to relinquish its position unless some conditions are satisfied. For example, you may want to verify a text entry before you allow the user to switch to another field. If `AllowTargetSwitch()` returns true, a switch is allowed. Otherwise, the target object cannot change.

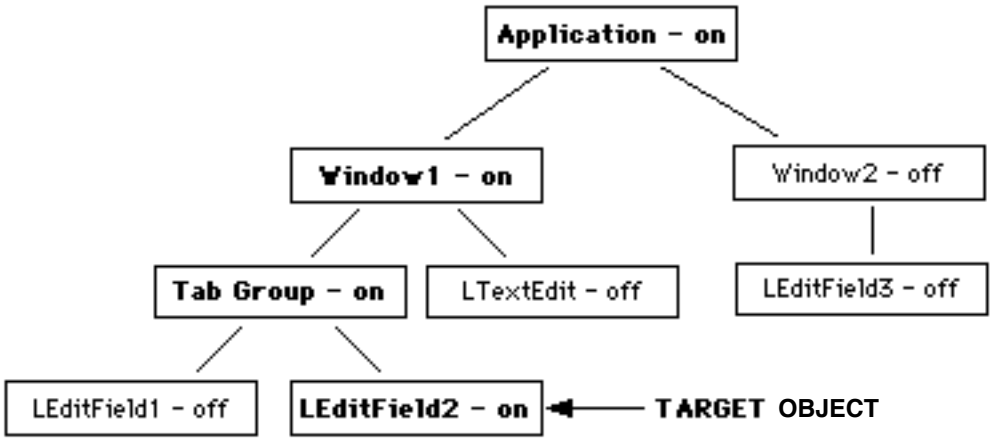
If you want to take some action when a commander becomes the target object or stops being the target object, override the member functions `BeTarget()` and `DontBeTarget()`. For example, you may want to highlight the target object, enable other objects, or per-

form setup duties when an object becomes a target. Another example is LListBox and its use of LFocusBox.

Duty Handling

An application may have multiple branches in the command hierarchy. Only one branch is active or "on duty" at any given moment, and that's the branch that ends at the target. When the target object changes, the new target's chain of command (from the target object back up to the application) returns to duty. The previous target object's chain goes off duty. Figure 10.2 illustrates the duty concept.

Figure 10.2 The duty property of commanders



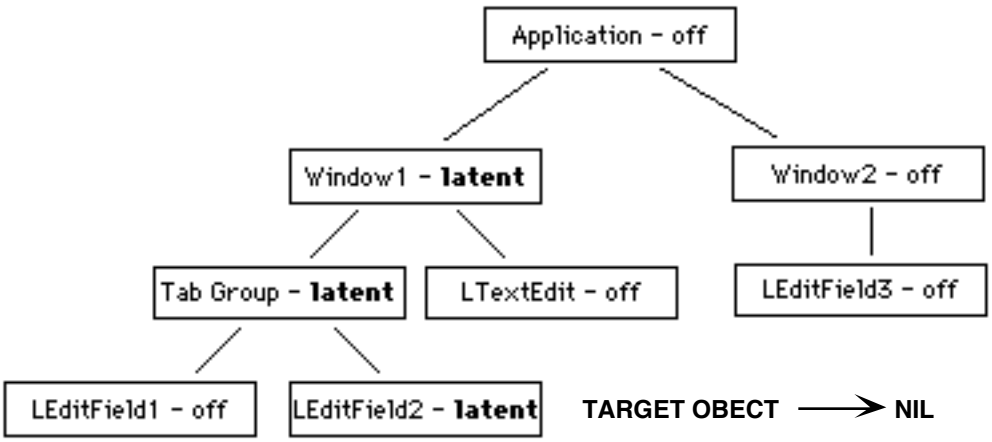
The target is not necessarily the lowest item in the command hierarchy. When a commander becomes the target object, its subcommanders (if any) are not automatically put on duty. Duty flows upward from the target object to the topmost commander. For example, if Window1 in Figure 10.2 were the target object, then all the commanders lower than it in the command hierarchy would be off duty.

If there are multiple command chains, the user can switch chains. In fact, if the application moves to the background all chains go off duty and the target object is set to nil. A background application cannot receive menu commands or keystrokes, so there can be no target object.

What happens when the application becomes active again? For example, look at Figure 10.2. Imagine the application has been suspended. When it resumes, LEditField2 should be restored as the target object.

To make that possible, PowerPlant allows each commander to have one of three states: on duty, off duty, or **latent**. If a commander is latent, it means that if the chain as a whole were on duty, the particular commander would also be on duty. Figure 10.3 shows the same command hierarchy as in Figure 10.2, but with the application suspended. As a result, certain commanders are now latent rather than on duty.

Figure 10.3 The duty property while suspended



When an application resumes or a chain within the application is activated (for example, by switching windows), PowerPlant searches through the command hierarchy for latent subcommanders and puts them back on duty. The deepest latent subcommander becomes the target object.

Now that you have the duty concept in hand, let's look at the functions each commander has for managing its duty status.

Table 10.3 LCommander duty handling functions

Function	Purpose
`IsOnDuty()`	return duty state
`GetLatentSub()`	return the latent subcommander
`SetLatentSub()`	specify the latent subcommander
`PutChainOnDuty()`	put command chain on duty
`PutOnDuty()`	called when this commander is going on duty
`TakeChainOffDuty()`	take command chain off duty
`TakeOffDuty()`	called when this commander is going off duty

When you create a window, and you want one of its commander panes to be the "default" pane that becomes active when that window becomes active, use `SetLatentSub()`. A commander may have no more than one latent subcommander.

If you want something to happen when a commander goes on or off duty, override the `PutOnDuty()` and `TakeOffDuty()` functions. For example, you might want to outline a frame when on duty, and hide a frame when off duty.

TIP: Being on duty is not the same as being the target object. The target object is always on duty. An on-duty commander is not always the target object. If you want something to happen when a commander becomes the target object, override `BeTarget()` and `DontBeTarget()`.

Command and Keystroke Handling

Fundamental to any commander's behavior is its ability to handle commands and keystrokes. A command is a menu selection or command-key equivalent. A keystroke occurs when the user types a

key. Whenever one of these two events occurs, control passes to the target commander.

Each commander has three principal functions to handle the command or keystroke, as listed in Table 10.4.

Table 10.4 Some LCommander command and key handling functions

Function	Purpose
`ObeyCommand()`	respond to a menu command
`FindCommandStatus()`	enable, disable, or mark a menu item
`HandleKeyPress()`	respond to a key event

You override these three functions regularly in PowerPlant applications. In fact, it is your definition of these functions that is responsible for much of the unique behavior of your own application.

You set up your own menus, your own menu commands, and define how your application or its component parts respond to each command. When the user chooses a menu item, the target commander's `ObeyCommand()` function receives an identifying command number that corresponds to a menu item. You respond accordingly.

You override the `FindCommandStatus()` function to enable, disable, and/or mark menu items as appropriate for your target object.

When the user types a key, the target commander's `HandleKeyPress()` function receives the event and responds accordingly.

Because the flow of duty is upward in the command hierarchy, the target object gets first crack at responding to commands and keystrokes, and at setting up the menus appropriately. If the target object changes, the new target object can do things differently. We discuss how all this works in the next section on menus.

Making and Managing Menus

To make your application work, you will need menus. In this section we discuss:

- Menu Strategy—how PowerPlant implements menus.
- Menu-Related Resources—the application resources you need for creating and using menus.
- Command Numbers—how the command numbering system works in PowerPlant.
- Adding Menus—how to add a menu to a PowerPlant application.
- Responding to Menu Commands—how to identify and respond to the user's menu choices.
- When To Update Menus—how to mark and modify menu items as necessary.
- Working With LMenuBar and LMenu—specific functions you may need.

Let's start with an overview of PowerPlant's menu strategy.

Menu Strategy

In traditional Macintosh programming, when the user chooses a menu item you dispatch control based on the menu ID and the item number for that particular item and menu. The Macintosh Toolbox routine `MenuSelect()` returns a number that contains both pieces of information.

As a result, traditional Macintosh menu dispatch code is heavily dependent upon item position. If you change the position of a command in a menu, or you move a command from one menu to another, you must rewrite the routine that dispatches menu choices.

To eliminate this problem, PowerPlant associates each menu item with a unique command number. A PowerPlant application maintains a "map" that says, for example, "The fifth item of the Edit menu has a command number of 14." The map contains a command

number for every item of every menu in the application. (Actually, there's an exception to this that we'll discuss in just a bit.)

The "map" is really a series of Mcmd resources, one for each MENU resource in the application.

When the user chooses a menu item, PowerPlant looks up the associated command number in the Mcmd resource, then sends the command number to the target commander's `ObeyCommand()` function. Because you know what each command number represents, you know the identity of the menu item the user chose.

If you decide to reorganize your application's menus, you do not have to modify your menu parsing and dispatch code. The command number remains unchanged. You must, however, modify your Mcmd resources so the "map" remains consistent with the location of menu items. If you use Constructor to build and modify your MENU resources, it takes care of this for you.

NOTE: What's really happening here is that the designers of PowerPlant know that developers change their menus. So PowerPlant moves responsibility for tracking position changes out of code and into a resource. You must still modify something if you reorganize a menu. With PowerPlant you modify the Mcmd resource rather than your source code.

Menu-Related Resources

PowerPlant uses three resource types to manage menus:

- MBAR
- MENU
- Mcmd

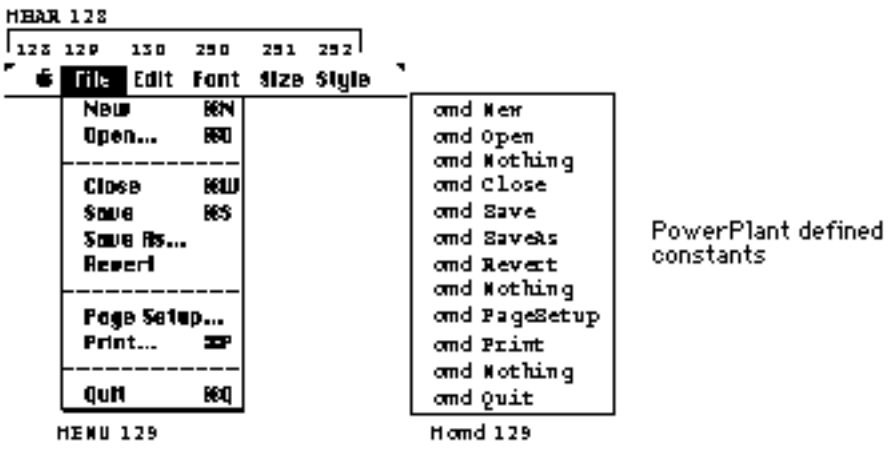
You can use Constructor to create all three resources.

The MBAR and MENU resources are standard Mac OS resources for defining the contents of the menu bar, and an individual menu re-

spectively. PowerPlant uses them in exactly the same way that any Macintosh program would.

The Mcmd resource is the "map" that tells PowerPlant what command number to associate with each menu item. There is one Mcmd resource for each MENU resource. The resource ID number for the Mcmd resource must be the same as the associated MENU resource. If you use Constructor, it takes care of the Mcmd numbering for you. Figure 10.4 illustrates the relationships between the MBAR, MENU, and Mcmd resources.

Figure 10.4 Example of Menu-related resources



If you use PowerPlant stationery to build a new project, the file `<PP Starter.ppob>` contains the MENU and Mcmd resources necessary to support the standard Apple, File, and Edit menus. The file `<PP Starter.rsrc>` contains the MBAR resource. You can use these as a starting point for further development.

See also "Installing Resource Templates" on page 61 for information on installing Mcmd resource templates.

Command Numbers

Each menu command to which you respond must have a unique number. PowerPlant reserves command numbers –999 to 999 for its own use.

For example, command number 0 (zero) is reserved for a command that does nothing. The standard menu commands for the Apple, File, Edit, and font-related menu items are defined in `PP_Messages.h`. Several of the enumerated constants are listed in Figure 10.4.

When you create an Mcmd resource, you assign a command number to each menu item. You are free to use any unique number as long as it isn't in the reserved range. However, using a systematic approach of some sort when assigning command numbers can help you avoid errors and maintain consistency.

In an Mcmd resource, assign zero as the command number for a separator bar in the associated menu. When using other PowerPlant-defined values, refer to `PP_Messages.h` for the value to use.

Negative command numbers

Under typical circumstances, you should use positive numbers for your menu commands. PowerPlant has facilities for enabling, disabling, marking, and changing the text for menu items. You cannot use regular PowerPlant techniques to do any of these things if the command has a negative number.

Some negative command numbers have special meaning in PowerPlant, including synthetic commands. We will revisit negative command numbers when we discuss dialogs in Chapter 12.

Synthetic command numbers

So far we have overlooked one significant problem with the PowerPlant menu strategy. What do you do for menus whose contents are defined at runtime, such as the Apple menu or the Font menu? You cannot know what items will be in these menus, so you cannot put entries in an Mcmd resource for them.

In a PowerPlant application, you build such a menu at runtime in the traditional way. We will use the standard Apple menu and a hypothetical Font menu as examples in this discussion.

The MBAR resource for the application has the resource ID numbers of all the MENU resources. The MENU resource for the Apple menu has the menu title (the Apple symbol) and the first About item. The MENU resource for a Font menu would have the menu title and no items.

The PowerPlant LMenuBar constructor builds the Apple menu using traditional techniques, as shown in Listing 10.1

Listing 10.1 Building the Apple menu

```
MenuHandle macAppleMenuH = ::GetMenuHandle(MENU_Apple);
if (macAppleMenuH != nil) {
  ::AppendResMenu(macAppleMenuH, 'DRVR');
}
```

The Apple menu has an Mcmd resource that contains an entry for the About item, because that is a standard item in the Apple menu. You cannot have an Mcmd entry for other items in the Apple menu. Similarly, you cannot have entries in an Mcmd resource for items in a font menu.

When the user chooses a menu item that does not have an associated entry in an Mcmd resource, PowerPlant generates a *synthetic command number* for that item.

A synthetic command number is a 32-bit number. The high 16 bits contain the menu ID, the low 16 bits contain the menu item. This is just like the return value from the traditional `MenuSelect()` Toolbox function, with one exception. The synthetic command number is negative.

After creating the synthetic command number, PowerPlant passes that command number to you, just like it would for a regular command number retrieved from an Mcmd resource.

Using synthetic command numbers

When you receive a menu command, you don't know if it is from an Mcmd resource. It might be a synthetic command. You call `LCom-`

`mander::IsSyntheticCommand()` to determine if the menu command is synthetic or not. If the command is a synthetic command, the function returns true and supplies the menu ID and item number for the chosen menu item.

You can then process the command properly, because you have the required information: menu ID and menu item.

For example, Listing 10.2 shows how `LApplication::ObeyCommand()` responds to synthetic commands from the Apple menu.

Listing 10.2 Handling synthetic command numbers

```
Boolean
LApplication::ObeyCommand(CommandT inCommand, void *ioParam)
{
  ResIDT        menuID;
  Int16        menuItem;

  // check for synthetic command
  if(IsSyntheticCommand(inCommand, menuID, menuItem))
  {
    if (menuID == MENU_Apple)
    {
      Str255 appleItem;
      ::GetItem(GetMHandle(MENU_Apple), menuItem, appleItem);
      ::OpenDeskAcc(appleItem);
    } else {
      cmdHandled = LCommander::ObeyCommand( inCommand, ioParam)
    }
  } else { // non-synthetic commands
    switch (inCommand) {

      case cmd_About:
        ShowAboutBox();
        break;
  ...
```

Note that the About Box has a non-synthetic command number. If the command number is not synthetic, then you simply test for the

command number directly—typically in a `switch` statement. There is no need to use the menu ID or menu item number for a regular command. In fact, that data is not readily available.

Now that you understand the strategy and details behind PowerPlant menu management, let's talk about how you accomplish real menu-related tasks.

Adding Menus

Adding a menu to your application is simple.

1. Create a MENU resource.
2. Create an Mcmd resource with the same ID number as the associated MENU resource. Enter command numbers for each item in the menu.
3. Modify the MBAR resource by adding the ID number of the new MENU resource to the MBAR resource.

If you use Constructor, this process is even simpler. You simply create a menu bar resource, add the necessary menus, and specify the menu items and command numbers. Constructor creates and manages the resources automatically. If you move menu items around, Constructor keeps the Mcmd resource updated. See the Constructor manual for details of menu editing in Constructor.

WARNING! If you don't use Constructor and you modify the position of menu commands in the MENU resources, you must modify the associated Mcmd resources as well. Failure to do so will cause unpredictable results.

Menus are created as part of the default `LApplication::Run()` function, which creates the menu bar (and all menus on the menu bar as specified in the application resources). However, there are times when you must specify a menu at runtime, and cannot do so in advance. For example, you have to set the contents of the Font menu at runtime because you cannot know in advance what fonts are available on any particular machine.

Use the `Initialize()` method of your application class to accomplish this work. The default function in the LApplication class is empty. Override this function in your own LApplication class to do any additional menu setup work that's required for your application. The default `LApplication::Run()` function calls `Initialize()` as part of the application setup process, so your function will be called before you have to start handling events. The code exercise for this chapter demonstrates the technique. You'll see it used in some other exercises as well.

Responding to Menu Commands

Under most circumstances, the default PowerPlant behavior is all you need for menu dispatch. PowerPlant identifies the menu choice, retrieves the associated menu command, and passes the command to the target commander. It is the target object's responsibility to handle the command or not.

The calling chain for this dispatch is as follows. When a click occurs in the menu, `LEventDispatcher::ClickMenuBar()` (in the application object) gets the command number. It then calls the target commander's `ProcessCommand()` function. The `ProcessCommand()` function passes the command on to attachments, and then calls the target commander's `ObeyCommand()` function.

`ObeyCommand()` is where command identification and response occurs. It is also the only function in this entire dispatch series that you are likely to override.

A typical `ObeyCommand()` function will test for synthetic commands, if they are used in your application and are of significance to the particular commander. For example, a text-related object would be very interested in a synthetic command from a Font menu. It calls `IsSyntheticCommand()` to determine if the command is synthetic, and to get the menu ID and item number. After you identify menu and item, you respond appropriately. Our hypothetical text commander might change the font it uses when displaying its contents, for example.

The typical `ObeyCommand()` function has a `switch` statement with a `case` for each command of interest to the object. In response to the command, you do whatever is appropriate for your application.

Finally, the typical `ObeyCommand()` function calls its inherited `ObeyCommand()` function for any commands it does not handle. Otherwise you won't get the benefit of command testing and response in the base class.

Although we discuss windows in the next chapter, they make a great example here. LWindow is a commander. If you derive your own window class from LWindow, your class's `ObeyCommand()` function should call `LWindow::ObeyCommand()` for any command it does not handle directly. This way you get to take advantage of LWindow's code.

LWindow, in turn, inherits directly from LCommander. If `LWindow::ObeyCommand()` does not handle the command, it calls `LCommander::ObeyCommand()`. This call gets the supercommander and calls the supercommander's `ProcessCommand()` function. This gives the supercommander the opportunity to act or pass on the command.

This approach means that a commander is not required to handle every possible command. Each commander handles the commands for which it is responsible. If it cannot handle the command, it passes responsibility back up the command chain to a higher commander. This is the code-level implementation of the bottom-up command chain hierarchy we talked about in the chapters on application framework design and PowerPlant design.

If the user issues a quit command, for example, it is likely that the command would filter up through the command chain all the way back to the application. Remember, the application is itself a commander, so it has an `ObeyCommand()` function. Because quitting is really an application-level chore, in `LApplication::ObeyCommand()` you'll find this code:

```
case cmd_Quit:
  SendAEQuit();
  break;
```

No other object needs to know how to handle a quit command, the application object takes care of it.

When To Update Menus

As a Macintosh programmer you know that you must update the appearance of menu items as well as respond to menu choices. Before we discuss *how* to update menu items, let's talk about *when* PowerPlant updates menu items, and how to control when a menu updates.

There are two common strategies for updating menus: update before displaying a menu, and update whenever an event occurs that might change the state of a menu. You can think of these approaches as "update before display" and "update as needed."

Update before display

In the first strategy, you update menus only when the user clicks in the menu bar or types a command key. Before displaying the menu, you update it so the menu's contents reflect the current state of the program. The state of a menu while it is not displayed is really unimportant. The only time it is important that the menu's contents accurately reflect the state of the program is when you actually look at the menu.

The difficulty with this strategy is that one part of a menu is always visible—the menu title in the menu bar. If all of a menu's items are disabled, the menu title should also be dimmed. Assume that as the user works with a program, such a situation arises—that is, the state of the program is such that all of a menu's items would be disabled. The menu title does not reflect that situation, because the menu hasn't been updated. The user clicks in the menu bar, and menus update. The user suddenly finds that a menu that appeared enabled turns out to be disabled! This is bad human interface design.

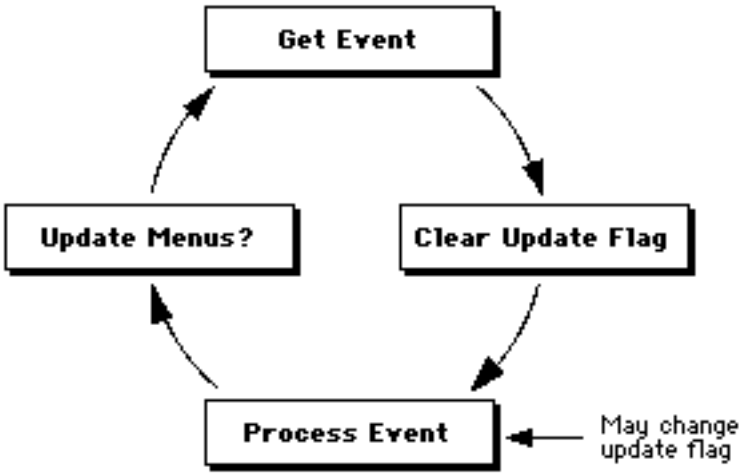
One solution would be to have a secondary mechanism for updating the menu bar whenever an event occurs that changes a menu's title. To do that reliably, you have to track every event that might cause a change in any menu item's state. If you do that, you will find yourself implementing PowerPlant's menu updating strategy.

Update as needed

PowerPlant takes the second approach to menu updating. PowerPlant keeps menu items current at all times. As a result, the menu bar accurately reflects the state of the application.

PowerPlant maintains a flag that reflects the state of menus. If the flag is set, menus are considered "dirty" and in need of updating. After processing each event, if the flag is set PowerPlant updates menus. Figure 10.5 illustrates the process.

Figure 10.5 PowerPlant menu updating logic



PowerPlant retrieves an event. It clears the menu update flag and processes the event. It is important to note that the event might be a click in the menu bar, in which case a menu is displayed *without additional updating*. PowerPlant relies on the fact that menus are always current.

When an event occurs that might change a menu (such as a click in the window content), PowerPlant sets the menu update flag. After

the event is fully processed, if the update flag is set then PowerPlant updates menus. It calls the target commander's `FindCommandStatus()` function to do this. We'll discuss what to do in this function in just a bit.

The events that cause PowerPlant to update menus are:

- switching the target
- click in the window content
- click in the menu bar
- command key press for a menu item
- activate event
- resume event
- Apple event
- modifying a document
- posting an undo action

Note that a click in the menu bar causes a menu update only after the menu is displayed, the user makes a choice, and the resulting command is fully processed.

An occasion may arise when you want to force a menu update as a result of some other occurrence. For example, you may want to enable the Save item in the File menu in a text processor only when there is text in the window. To force a menu update, call `LCommander::SetUpdateCommandStatus()`. Pass true as the only parameter. This call sets the menu update flag. When control returns to the main event loop, PowerPlant will update the menus.

Updating Menu Items

When it is time to update menu items, PowerPlant goes through each item in each menu and gets a command number. What happens depends upon the command number. There are five possible results:

- The command number is positive.
- The command number is zero.

- The command number is -1.
- The command number is negative.
- The command number is synthetic.

The command number is positive

We call positive menu commands non-synthetic commands to distinguish them from synthetic commands. These are the most common command numbers for menu items.

The target commander's `FindCommandStatus()` function will be called once for each menu item that has a positive, non-zero menu command. Like the `ObeyCommand()` function, `FindCommandStatus()` receives the command number that identifies the menu item involved.

In response, your commander determines what the status of that menu item should be (based on application context), and provides the necessary information to the caller. `FindCommandStatus()` has a series of parameters, as listed in Table 10.5.

Table 10.5 FindCommandStatus() parameters

Data Type	Name	Purpose
`CommandT`	`inCommand`	command number
`Boolean&`	`outEnabled`	provide true if enabled
`Boolean&`	`outUsesMark`	provide true to use a mark
`Char16&`	`outMark`	provide the mark to use
`Str255`	`outName`	provide the menu item text

Each of the output parameters has a default value set before entry into `FindCommandStatus()`. By default, a menu item is disabled, unmarked, and the text remains unchanged.

If you want the item enabled, set `outEnabled` to true.

If you want a mark to appear before the menu item (like a check mark, dash, diamond, or blank space to clear a mark), set `outUsesMark` to true. If you use a mark, you must also specify the mark you want to use in `outMark`.

To clear a mark, set `outUsesMark` to true and `outMark` to zero. Setting `outUsesMark` to false does *not* clear an existing mark.

If you want to modify the menu item text, set `outName` to the text you want to use. If you do not want to modify the item text, do not modify this parameter.

Each commander's `FindCommandStatus()` function handles those menu items for which the commander is responsible. If the commander's `FindCommandStatus()` function receives a command it does not recognize, it should pass the request on to its inherited `FindComandStatus()` function. This process works just like the `ObeyCommand()` design.

Call the `FindCommandStatus()` function inherited from your base class to get the benefit of the code in the base class. If the base commander from which you inherited is LCommander, then call `LCommander::FindStatus()`. This passes the request up to the supercommander.

The command number is zero

If the command number is zero, PowerPlant does nothing. The number zero indicates a menu item that is always disabled, like a separator bar item in the menu. There is no need for you to deal with such an item. It is and always remains disabled.

The command number is -1

The -1 number is a special command number. When PowerPlant encounters a command number -1 (`cmd_UseMenuItem`), it generates a synthetic menu command for this item, and then calls the target commander's `FindCommandStatus()` function, just like it does for positive menu commands.

The `FindCommandStatus()` function should deal with this item in exactly the same way that it handles non-synthetic items. However, to identify the menu item you must call `IsSyntheticCommand()` to get the menu ID and menu item number. Then you can determine how to update the item based on application context.

The command number is negative

PowerPlant does not update items with negative command numbers (except for the -1 command number). Items with a non-synthetic, negative command number remain in the same state, typically enabled. The target commander's `FindCommandStatus()` function is not called for these items.

However, there is a loophole that allows you to modify menu items with negative command numbers, as you'll see when we discuss updating items with synthetic command numbers.

The command number is synthetic

When PowerPlant searches for a menu item's command number, there may be none at all. If there is no command number, PowerPlant generates a synthetic command number.

PowerPlant does not update these items. The target commander's `FindCommandStatus()` function is not called for items with synthetic command numbers.

Clearly this presents a problem. You may need to update a menu item that has a synthetic command. For example, you may want to put a check mark in front of the current font in a Font menu.

PowerPlant provides a mechanism for you to accomplish that task, and other updates of synthetic menu items.

In addition to calling `FindCommandStatus()` once for each positive menu command, PowerPlant also calls the target commander's `FindCommandStatus()` function *once for each menu as a whole*. Here's how it works.

After calling `FindCommandStatus()` for each eligible menu item, PowerPlant manufactures a synthetic command for item zero of

each menu—that is, for the menu title. PowerPlant then calls `FindCommandStatus()` with this synthetic command number. This gives the target object an opportunity to do something to the menu as a whole.

Note that this call is made once for each menu, regardless of whether the menu has synthetic commands, so that you can operate on an entire menu. However, this is your only opportunity to modify synthetic menu items.

For example, assume you have a text-processing application with a Font menu. Assume also that it is menu update time. Your text target object's `FindCommandStatus()` function *is not* called for any item in the Font menu, because they all have synthetic (negative) command numbers.

However, the text object's `FindCommandStatus()` function *is* called once for the Font menu as a whole. The command number is a synthetic command number for item zero of the menu. Your `FindCommandStatus()` function calls `IsSyntheticCommand()`. This call returns true and provides the menu ID for the Font menu, and zero for the item number.

At this time you can act on the items in the Font menu. For example, you may enable all of them, disable them, or put a check mark in front of the item for the current font. You do this using standard Macintosh Toolbox calls. The details, of course, are dependent upon your own application.

Working With LMenuBar and LMenu

Finally, you may have noticed that there hasn't been much talk about these two classes. That's because most of their utility is designed for internal PowerPlant use. You won't have to use objects of either class directly very often.

The exception to this rule is when you are dealing with menus like the typical Font menu. In that case, you need to get the Mac OS MenuHandle so that you can perform traditional Mac OS menu-management tasks like putting a check mark in front of the item

representing the current font. You also need to work with the Menu Manager if you dynamically alter menus at runtime.

An application has one LMenuBar object. You can always get a pointer to this object by calling the static function `LMenuBar::GetCurrentMenuBar()`.

You can use LMenuBar to add or remove menus dynamically, although doing so is not a recommended feature of the human interface. Use `LMenuBar::AddMenu()` to add a new menu at runtime. Use `LMenuBar::RemoveMenu()` to remove a menu at runtime. If the menu has submenus, they are added or removed as well.

Each menu in a PowerPlant application has an associated LMenu object. To get a pointer to the menu object of choice, you call the LMenuBar's `FetchMenu()` function. You provide the menu ID.

After you have a pointer to the menu object, getting the Macintosh Toolbox MenuHandle is simple. Call the menu object's `GetMacMenuH()` function. You now have a Mac OS MenuHandle, and you can do what you want with the menu.

However, LMenu provides some functions that are more useful in the PowerPlant context than using the Mac OS Menu Manager directly. They are listed in Table 10.6.

Table 10.6 Some LMenu functions

Function	Purpose
`GetMacMenuH()`	return Mac OS MenuHandle
`GetMenuID()`	return menu ID
`InsertCommand()`	insert an item with the specified text and command number into the menu
`RemoveCommand()`	remove the item (specified by command number) from the menu
`RemoveItem()`	remove the specified item from the menu

Function	Purpose
`SetCommand()`	set the specified item's command number
`ItemIsEnabled()`	return whether specified item is enabled

Finally, the LMenuBar and LMenu combination is another independent PowerPlant design element that you can use independently of the rest of PowerPlant. Simply include the necessary files in your project, and you're on your way. The command dispatch and menu updating mechanisms are an integral part of PowerPlant, and separate from the menu classes.

Summary

LCommander objects maintain the chain of command and duty in an application. The application has one target commander that receives all commands and keystrokes directly.

The target object can respond to a command, or pass it up the command chain. The target object also tells PowerPlant the status of menu items during menu updates.

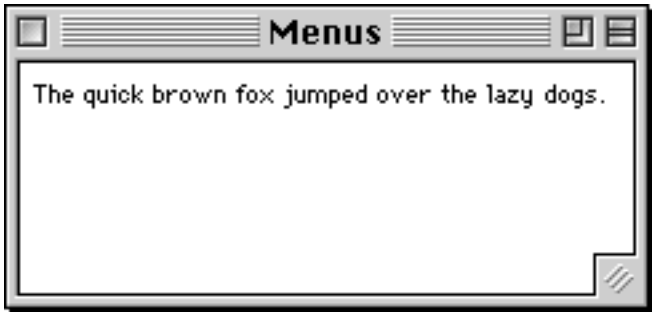
You build menus using MBAR, MENU, and Mcmd resources. PowerPlant associates a unique command number with each menu item, and uses that number when handling commands or menu updates. PowerPlant handles all menu dispatch. You override the `ObeyCommand()`, `FindCommandStatus()` and `HandleKeyPress()` functions in your commanders to implement your application's behavior.

As usual, now it's time to put all this to work in real code.

Code Exercise

This exercise introduces you to commanders, and to menus. You create a simple application named "Menus." This application displays a simple window, as shown below in Figure 10.6. You can create an arbitrary number of windows.

Figure 10.6 The Menus application



Each window contains a caption. Unlike the standard LCaption object, however, this caption is dynamic and a commander. It responds to menu commands to change font, size, and style. You cannot change the text in this object using the Menus application. It is a caption, not an editable text item.

Feel free to examine the PPob resource for this window. The CDynamicCaptionCmdr class is a custom class. The class ID is DyCC. In addition to the usual caption information, this item stores the menu resource ID for the font and size menus, and the item numbers of the first and last size items.

Because this exercise concentrates on menus, in this exercise we won't explore the visual interface further. We will explore the menu resources. And then you write the code to identify and respond to commands, and to update menus.

The Menu Resources

You can use Constructor to examine the menu-related resources. They are in the file `Menus.ppob`. You won't modify any of these resources, but you should take a look at them so you understand what PowerPlant requires for menus.

Examine the MENU resources. In addition to the standard Apple, File, and Edit menus, there are Font, Size, and Style menus. Notice the MENU resource ID numbers for these menus. They are 250, 251, and 252 respectively. These menus are very common in Macintosh applications, and the `PP_Messages.h` file defines these constants:

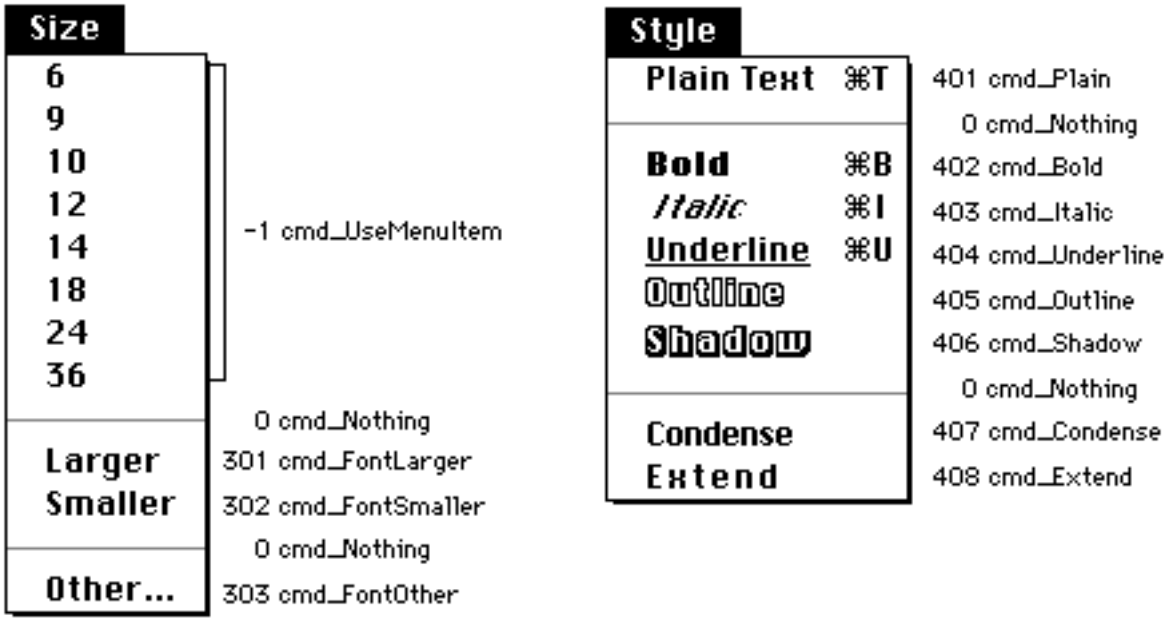
```
const MessageT cmd_FontMenu = 250;
const MessageT cmd_SizeMenu = 251;
const MessageT cmd_StyleMenu = 252;
```

The Font menu has no items. The Size and Style menus have the standard items you typically see in these menus. Close the MENU resources when you're through with them.

Examine the MBAR resource. Notice that it contains an entry for each of the six menus. PowerPlant will build the menu bar containing each of these menus. The combination of MENU and MBAR resources is all that's necessary to make a menu appear in a PowerPlant application. To make that menu work, however, you typically use an Mcmd resource.

Constructor maintains the Mcmd resources automatically based on the command ID you set for each menu item. Figure 10.7 illustrates the relationship between the Size and Style menus and their respective commands.

Figure 10.7 Menu commands in the Menus app



There is one command for each of the 13 items in the Size menu. The first 8 items have a command number of -1. This is the `cmd_UseMenuItem` value we discussed earlier in this chapter. The other items in the Size menu and all the items in the Style menu have positive or zero command numbers. The corresponding PowerPlant constants for these standard items are defined in `PP_Messages.h`.

There are similar standard commands defined in PowerPlant and used in the Apple, File, and Edit menus. Feel free to examine them as well.

Notice that there is no Mcmd resource for the Font menu. PowerPlant generates synthetic commands for these items at runtime.

Implementing Menus

In this part of the code exercise you write the code to populate the Font menu, establish a command hierarchy, recognize and respond to menu commands, and handle menu updating. In the process you will work with both kinds of commands, synthetic and non-synthetic. You also see the practical difference between a pure synthetic command and the `cmd_UseMenuItem` command.

Before we get started, a little context will help. Let's look at the dynamic caption object and how it is implemented. The code for the dynamic caption object is provided for you.

1. **Examine CDynamicCaption.**

   `class declaration` `CDynamicCaption.h`

   Take a look at the class declaration for CDynamicCaption. It inherits from LCaption, and therefore has all the features of a regular LCaption object. In addition, the CDynamicCaption object has a text traits record. Most of the public functions of CDynamicCaption are sim-

ple accessors for the text traits record or one of its members such as font, size, style, and justification.

The `FinishCreateSelf()` routine simply sets the `mTextTraits` member.

The `DrawSelf()` routine gets the current text traits, sets the port characteristics to match, and calls `UTextDrawing::DrawWithJustification()` to draw the contents.

Based on this design, when you respond to a selection that affects the text traits, all you have to do is modify the text traits information and refresh the pane. If you look at the `Set...()` functions among the accessors, you'll see that they do just that. They set the appropriate text traits field, and call `Refresh()`. As a result, the caption draws itself with the new settings.

Close the file when you are finished.

NOTE: A CDynamicCaption object is never instantiated in this application. CDynamicCaption is used as a base class for CDynamicCaptionCmdr, discussed in the next step.

---

2. Examine CDynamicCaptionCmdr.

`class declaration` `CDynamicCaptionCmdr.h`

Left to itself, the CDynamicCaption object cannot respond to menu commands, because it is not a commander. In a simple application like this, you could design the application object to keep track of the CDynamicCaption object, recognize and respond to each menu command, set the CDynamicCaption object text traits, and so forth.

However, it is usually wiser to distribute responsibility for commands to the objects that respond to the commands. In order to accomplish that design, the object must be a commander.

CDynamicCaptionCmdr solves the problem. It inherits from CDynamicCaption and LCommander. It is a dynamic caption object that

can respond to commands. The PPob resource for the window in this application specifies a CDynamicCaptionCmdr object.

The only functions this class overrides are `ObeyCommand()` and `FindCommandStatus()`. You will write those functions later in this exercise. Before you get that far, you must do some setup work.

Close this header file when you are finished.

3. **Register custom classes**

`CMenusApp()` `CMenusApp.cp`

This application uses two custom pane classes. Although you only instantiate CDynamicCaptionCmdr, you should register both. The CDynamicCaptionCmdr object inherits from CDynamicCaption. PowerPlant will create the CDynamicCaption object as part of the CDynamicCaptionCmdr object.

The existing code sets up debugging and registers PowerPlant classes. Register both custom classes.

```
// Register required PowerPlant core class.
  RegisterClass_(LWindow);

// Register custom classes.
RegisterClass_( CDynamicCaption );
RegisterClass_( CDynamicCaptionCmdr );
```

4. **Populate the Font menu.**

`Initialize()` `CMenusApp.cp`

The content of the Font menu depends upon the runtime environment. You must fill in the items as the application starts up. You perform final setup work in the application's `Initialize()` function.

You should get the menu bar object, get the Font menu from it, get the Mac OS menu handle, and call `::AppendResMenu()` to add resources of type FONT.

```
// Setup the font menu.
LMenuBar *theMenuBar = LMenuBar::GetCurrentMenuBar();
ThrowIfNil_( theMenuBar );
```

```
LMenu *theFontMenu = theMenuBar->FetchMenu( rMENU_Font );
ThrowIfNil_( theFontMenu );

::AppendResMenu( theFontMenu->GetMacMenuH(), 'FONT');
```

Unlike most of the other applications you have built so far, there is no code to make a window. This application supports multiple windows. At launch time, if an open-application Apple event is received, the `Startup()` function calls `ObeyCommand()` to create a new window. You write `ObeyCommand()` in the next step.

5. **Create a window.**

`ObeyCommand()` `CMenusApp.cp`

One command the application object recognizes is `cmd_New`. In response you should create a new window. The existing code identifies the message in a `case` statement. The defined constant for the PPob resource is `rPPob_MenusWindow`. Call the LWindow class creator function.

```
case cmd_New:
{
  // Create the window.
  LWindow *theWindow;
  theWindow = LWindow::CreateWindow(rPPob_MenusWindow, this );
  Assert_( theWindow != nil );
```

6. **Establish the latent command hierarchy.**

`ObeyCommand()` `CMenusApp.cp`

The command hierarchy for this simple application is (from top to bottom): application, window, caption. When you call the LWindow class creator function, you specify the application object as the commander. When the window activates, you want the caption object to become the target object. Therefore, you must make it the latent subcommander of the window. Then show the window.

To accomplish this task, now that you have the window, get a pointer to the caption object, and make it the latent subcommander

of the window. This code goes right after the code you wrote in the previous step, as part of the response to `new_Cmd`.

```
  Assert_( theWindow != nil );

// Get the caption.
CDynamicCaptionCmdr *theCaption;
theCaption = dynamic_cast <CDynamicCaptionCmdr *>
       (theWindow->FindPaneByID( kDynamicCaptionCmdr ));
Assert_( theCaption != nil );

// Make it the latent subcommander of window.
theWindow->SetLatentSub( theCaption );

// Show the window.
theWindow->Show();
```

The last line that shows the window is redundant if the window's visible attribute is set.

Notice that the default `case` passes unhandled commands on to the inherited `ObeyCommand()` function.

7. **Update the New item in the file menu.**

`FindCommandStatus()` `CMenusApp.cp`

The application object is responsible for the New item, both to respond to the command and update the menu. This is a non-synthetic menu command. The existing code identifies the case. The New item should always be enabled. Enable the menu item.

```
case cmd_New:
{
  // Enable the New command.
  outEnabled = true;
}
break;
```

Save your work and close the file. You have completely implemented the application object's behavior. The responsibility for other commands rests with the caption. In the following steps you

write the `ObeyCommand()` and `FindCommandStatus()` functions for the caption.

8. **Respond to synthetic commands**

`ObeyCommand()` `CDynamicCaptionCmdr.cp`

The caption may receive synthetic commands from either the Font menu or the Size menu. To complete this step you perform the following tasks.

a. Determine if the command is synthetic.

If the command is synthetic continue. Otherwise, you'll skip to the code that handles non-synthetic commands. You write that code in the next step.

b. Determine if it is a Font menu command.

If it is, continue. If it is not, you'll skip to the code that handles the Size menu. You write that code in substep d in this step.

c. Process the Font menu command.

Get the Font menu object, read the text of the item, and call `SetFont()` to update the text traits record. This completes handling of a Font menu command.

d. Determine if it is a Size menu command.

There are other synthetic commands besides the Font and Size menus. If this command is from the Size menu, continue. If it is not, you skip to the code that handles other synthetic commands. You write that code in substep f in this step.

e. Process the Size menu command.

Get the Size menu object, read the text of the item, convert it to a number, and call `SetSize()` to update the text traits record. This completes handling of a Size menu command.

f. Pass on other synthetic commands.

The caption object is not responsible for any other synthetic commands. If the command is synthetic, but not a Font or Size menu command, call the inherited `ObeyCommand()` function. In this case, that's `LCommander::ObeyCommand()`.

The solution code for this long step is listed here for reference.

```
Int16 theMenuItem;
// Is it a synthetic command.
if ( IsSyntheticCommand( inCommand, theMenuID, theMenuItem ) ) {
// Is it a Font menu command.
  if ( theMenuID == mFontMenuID ) {

    // Get the menu object.
    LMenu *theMenu;
    theMenu = LMenuBar::GetCurrentMenuBar()->
                FetchMenu( mFontMenuID );
    Assert_( theMenu != nil );

    // Get the menu item text - name of font.
    Str255 theFontName;
    ::GetMenuItemText( theMenu->GetMacMenuH(), theMenuItem,
                       theFontName );

    // Set the caption font.
    SetFont( theFontName );

  } else if ( theMenuID == mSizeMenuID ) {

    // Get the menu object.
    LMenu *theMenu;
    theMenu = LMenuBar::GetCurrentMenuBar()->
                FetchMenu( mSizeMenuID );
    Assert_( theMenu != nil );

    // Get the menu item text.
    Str255 theMenuText;
    ::GetMenuItemText( theMenu->GetMacMenuH(), theMenuItem,
                       theMenuText );

    // Get the size referred to by menu item.
    Int32 theSize;
    ::StringToNum( theMenuText, &theSize );

    // Set the caption size.
    SetSize( theSize );
```

```
  } else { // Neither Font nor Size menu.

    // Call inherited.
    cmdHandled = LCommander::ObeyCommand( inCommand, ioParam );

}
```

9. Respond to non-synthetic commands.

`ObeyCommand()` `CDynamicCaptionCmdr.cp`

The code in the previous step started with an `if` statement that called `IsSyntheticCommand()`. All the code in this step falls inside the `else` condition to that original `if` statement. In other words, if the command is synthetic, the code in the previous step handles it. Otherwise, the code in this step handles it.

We have given you most of the code in this step. The existing code sets up the `else` condition, switches based on the command number, and dispatches control to a variety of `case` statements. In this step, you complete two cases: `cmd_Plain` and the `default` case.

a. Respond to the Plain menu command.

Call the caption's `SetStyle()` function. Set the style to the Toolbox constant `normal`.

```
case cmd_Plain:
{
  // Set the caption style to normal (plain).
  SetStyle( normal );
}
break;
```

Existing code handles most other cases.

b. Pass on other menu commands.

The caption object is not responsible for other non-synthetic commands. Call the inherited `ObeyCommand()` function. In this case, that's `LCommander::ObeyCommand()`.

```
default:
{
  // Call inherited.
  cmdHandled = LCommander::ObeyCommand( inCommand, ioParam );
}
break;
```

You have now completely implemented commands in this application. Your final task is to update menu items. You accomplish that in the next two steps.

10. Update synthetic menu items.

`FindCommandStatus()` `CDynamicCaptionCmdr.cp`

Remember that the Font menu has no Mcmd resource. The Size menu does have an Mcmd resource, and assigns the value -1—`cmd_UseMenuItem`—to each of the items for setting a particular point size. PowerPlant behaves differently with respect to menu updating for items with a -1 command number, as you are about to see first hand in the code for this step.

Existing code calls `IsSyntheticCommand()`. Because the Font menu is populated completely by items with no Mcmd resource, PowerPlant does not call `FindCommandStatus()` on an item-by-item basis. Instead, PowerPlant calls this function once for the Font menu title, with an item ID of zero. The existing code tests to see if the Font menu is involved. If it is, you should:

a. Enable the Font menu title.

b. Get the name of the font used by the caption.

Use the CDynamicCaptionCmdr `GetFont()` function.

c. Get the menu object.

Get the menu bar object, use it to get the Font menu object.

```
if ( theMenuID == mFontMenuID ) {

  // Enable the menu title.
  outEnabled = true;
```

```
  // Get the name of the font used by caption.
  Str255 theFontName;
  GetFont( theFontName );

  // Get the menu object.
  LMenu *theMenu;
  theMenu = LMenuBar::GetCurrentMenuBar()->
              FetchMenu( mFontMenuID );
  Assert_( theMenu != nil );

  // Get the number of items in the menu.
```

The existing code uses Toolbox functions to count the number of items in the menu and set the proper mark for each item.

For the Size menu items that have command number -1, PowerPlant also creates a synthetic menu command. However, the `FindCommandStatus()` function is called for each item in turn. As a result, you can handle them individually. `FindCommandStatus()` is called once for the menu title as well.

The existing code identifies a synthetic command involving the size menu, and switches on the item. You should:

d. Enable the Size menu title.

If the item is the menu title, enable the item.

e. Process each other item.

All other size items are treated identically, so this can be a default case. You should enable the item. Then get the Size menu object. Existing code does the rest of the work.

```
switch (theMenuItem)
{
  case 0: // Menu title

    // Enable the menu title.
    outEnabled = true;
    break;

  default: // Other size items with -1 command
```

```
  {
    // enable the individual item
    outEnabled = true;

    // Get the menu object.
    LMenu *theMenu;
    theMenu = LMenuBar::GetCurrentMenuBar()->
                FetchMenu( mSizeMenuID ));
    Assert_( theMenu != nil );

    // Get the menu item text.
```

Existing code determines the size for the caption text. It uses Toolbox calls to match the size represented by the menu item with the text size. If they match, it sets a check mark. If they do not, the code sets no mark (thus clearing any previous mark).

Notice that the code for the Size menu does not loop. The Font menu code executes once only—for item zero, the menu title. The Size menu code executes repeatedly, once for each item with the -1 command number, and again for item zero—the menu title. In either case—Font or Size menu—the actual command number received by `FindCommandStatus()` is a synthetic command.

f. Pass on other menu commands.

You must pass any command you don't handle to the inherited `FindCommandStatus()` function. In this case, that's LCommander.

```
} else { // other synthetic command

  // Call inherited.
  LCommander::FindCommandStatus( inCommand,
    outEnabled, outUsesMark, outMark, outName );
}
```

You have now completely handled updating menu items for which PowerPlant generates synthetic menu items. Let's do the non-synthetic items.

11. Update non-synthetic menu items.

`FindCommandStatus()` `CDynamicCaptionCmdr.cp`

The code in the previous step started with an `if` statement that called `IsSyntheticCommand()`. All the code in this step falls inside the `else` condition to that original `if` statement. In other words, if the command is synthetic, the code in the previous step handles it. Otherwise, the code in this step handles it.

We have given you most of the code in this step. The existing code sets up the `else` condition, switches based on the command number, and dispatches control to a variety of `case` statements. In this step, you complete two cases: `cmd_Plain` and the `default` case.

a. Update the Plain menu item.

Enable the item. The item uses a mark, so set that flag as well. Then determine if the caption's current style is `normal`. If it is, use a check mark. If not, use no mark.

```
case cmd_Plain:
{
  // Enable the item, set the uses mark flag,
  // and get the mark.
  outEnabled = true;
  outUsesMark = true;
  outMark = (GetStyle() == normal) ? checkMark : noMark;
}
break;
```

Existing code handles most other cases.

b. Pass on other menu commands.

The caption object is not responsible for other non-synthetic commands. Call the inherited `FindCommandStatus()` function. In this case, that's from LCommander.

```
default:
{
  // Call inherited.
  LCommander::FindCommandStatus( inCommand, outEnabled,
                                 outUsesMark, outMark, outName );
```

```
}
break;
```

Of all the cases for which code is provided for you, the one worthy of note is the `case cmd_FontOther`. It loops to see if the caption text size matches any of the size items in the menu. If it does not, it modifies the text of the Other item to include the actual font size.

You have now completely implemented commands and menu updating. Save your work and close the file.

12. Build and run the application.

This has been a long exercise, but now its time to see the fruits of your labor. Make the project and run it. When you do, a window should appear. See Figure 10.6 on page 334.

Choose items in the **Font**, **Size**, and **Style** menus and watch how the caption responds. When you look at each menu, notice the check marks and how the menu updates properly. This is your code hard at work making sure the menu items are consistent with the state of the caption. The Other item in the Size menu is not supported, so it isn't checked even if the caption uses an "other" size.

Choose the **New** item in the **File** menu. A second window should appear, with another caption. Set its font, size, and style. The settings for this caption are independent of the caption in the first window. When you look in the menus, the correct item is checked for the active window and caption. Very cool.

Now, close both windows. What happens to the Font, Size, and Style menus?

Because there is no caption object to update these menus, they remain disabled (the PowerPlant default). In the design of this little application, that's appropriate behavior.

Continue exploring the command and menu updating process. If you would like to expand on this application, here are some suggestions.

- Enable the debugger and set breakpoints in the `FindCommandStatus()` and `ObeyCommand()` routines so you can watch the flow.

- Write code to implement the Other item in the Size menu. This will send you off into the world of dialogs, a topic not yet covered in this manual.
- Add a color menu to the application, and colorize the text. Perhaps you can use the color control from the Controls chapter.
- Enable the Font, Size, and Style menus when no window is open. Save the settings in a global default. When you create a new window, apply these settings to the window.
- Replace the caption with an editable text object.

Have a good time!

You have reached another milestone on the road to PowerPlant mastery. This exercise has given you practical experience working with all kinds of menu commands in PowerPlant.

We aren't completely through with menus just yet. In the next chapter you learn all about windows in PowerPlant. In the code exercise for that chapter, you'll build a Window menu that lists each open window. You'll do a lot more work with menus in that exercise.

11

Windows

LWindow is a complex class that descends from and adds a great deal to LView. You will use this class often in your PowerPlant programming, both directly and as a base class for your own window classes. It serves as a wrapper class for the Macintosh Toolbox WindowRecord structure, so you can easily create and use windows in PowerPlant.

The good news is, you already know 90% of what you need to know to use windows effectively. Chapter 7, "Views" introduced you to LView and its descendants, including LWindow. At the time we skipped any detailed discussion of the single most important kind of view, the window.

In this chapter we concentrate on those aspects of window objects that make a window different from other views. In our discussion we will cover three main topics:

- What is a Window—the window class and its descendant.
- Window Characteristics—the special features of a window that distinguish it from other views.
- Working With Windows—how to create and use windows in a PowerPlant application.

Along the way we will also encounter several window-related utilities in PowerPlant.

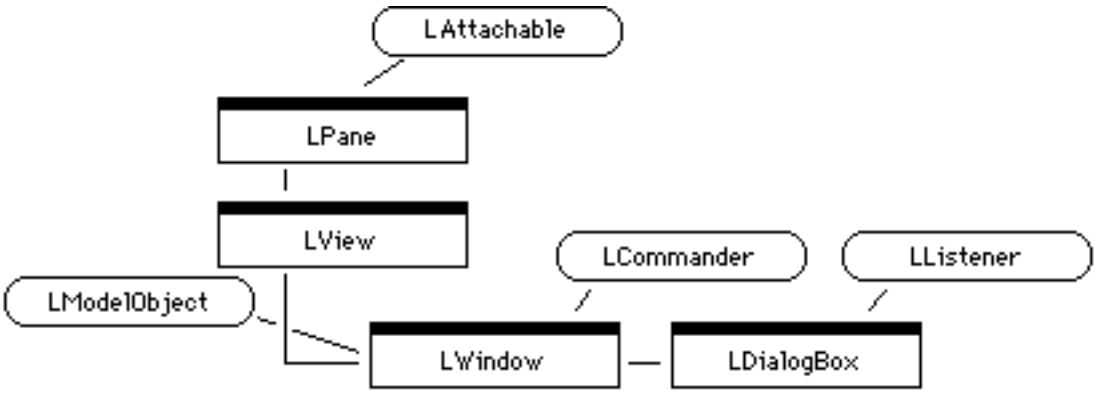
What is a Window

On the Macintosh, a window is the visual representation of a very special data structure that incorporates a GrafPort for drawing and other data to control drawing characteristics. On a monitor, this structure shows up as a window.

In PowerPlant, a window is an object of the LWindow class. This object is connected to a Mac OS WindowRecord, and appears on screen as an ordinary Macintosh window.

Figure 11.1 illustrates the inheritance chain for LWindow.

Figure 11.1 LWindow hierarchy



As you can see from the class diagram, LWindow has three direct ancestors.

First, LWindow inherits from LView, a descendant of LPane. That gives a window all the features of panes and views. Windows can receive and respond to mouse clicks within their bounds. They can contain other panes. Windows can perform all the other magic associated with panes and views, not the least of which is the ability to have attachments.

LWindow inherits from LCommander. That means that a window can receive and handle menu commands and key presses. It can update and manage menus as any other commander can.

Finally, LWindow inherits from LModelObject. Therefore, a window can respond to Apple events—it is scriptable.

LWindow has one descendant in PowerPlant, LDialogBox.

If you stop to think about all the important classes from which LWindow inherits, you see that its behavior encapsulates every-

thing we talked about in chapters 6, 7, and 9 on panes, views, and commanders. That's a lot of power.

Beyond that, LWindow has significant attributes and behaviors that are unique to it. In the rest of this chapter we discuss what they are.

See also Chapter 15, "Periodicals and Attachments," Chapter 12, "Dialogs," and *PowerPlant Revealed* for more on scriptability.

Window Characteristics

Do not lose sight of the fact that, although we mention this only in passing, *everything* you have learned about panes, views, and commanders applies to windows. That means that windows have an image, manage coordinate conversions, respond to clicks, and all the rest.

In addition to all those characteristics and behaviors, windows also have the following additional characteristics.

- Window Attributes—features PowerPlant uses to manage a window.
- Window Size and Zooming—the minimum, maximum, and standard dimensions of a window.
- Window Descriptor—how the window uses the descriptor characteristic of panes.
- Window Kind—how PowerPlant uses the Mac OS WindowRecord `windowKind` field.

We discuss additional window-related behaviors in "Working With Windows" on page 361.

Window Attributes

Each window has a set of special attributes that define the kind of window it is, and how the window behaves. These attributes determine the layer in which the window appears, the various controls that appear around the perimeter of the window (like a close box or zoom box), and some other aspects of window behavior.

Table 11.1 lists all the window attributes and their meaning. In the sections following the table we discuss each of these attributes in some detail.

Table 11.1 Window attributes

Attribute	Purpose
`windAttr_Modal`	in modal layer
`windAttr_Floating`	in floating layer
`windAttr_Regular`	in regular layer
`windAttr_CloseBox`	has close box
`windAttr_TitleBar`	has title bar
`windAttr_Resizable`	is resizable
`windAttr_SizeBox`	has grow box
`windAttr_Zoomable`	is zoomable and has zoom box
`windAttr_ShowNew`	show immediately when created
`windAttr_HideOnSuspend`	hide window when application is suspended
`windAttr_EraseOnUpdate`	erase before drawing
`windAttr_Enabled`	is enabled
`windAttr_Targetable`	is targetable
`windAttr_GetSelectClick`	process click that selects window
`windAttr_DelaySelect`	process click, then (perhaps) select window

Window layers

PowerPlant uses three layers to determine how a window behaves in certain respects. From front to back, the layers are:

- modal

- floating
- regular

Modal windows are always in front, and all other windows are inactive when a modal window is present. Modal windows must be dismissed before you can perform other actions in the program. The modal dialog is a perfect example. We'll discuss modal windows in Chapter 12, "Dialogs."

Floating windows are always active, except when a modal window is active. The floating layer is used for tool palettes and other kinds of accessory windows. These "float" above the regular front window. You may activate any regular window, and a floating window remains displayed in front of it. It appears to be active and frontmost, even though the regular window may be frontmost as far as the Mac OS is concerned.

As you'll see in the Working With Windows section, implementing floating windows is simple. Floating windows are one of the really nice features of PowerPlant.

Regular windows are beneath all modal and floating windows. The top regular window is active, except when a modal window is active. All other regular windows are inactive. Windows in the regular layer behave like normal Macintosh windows.

Peripheral window parts

The next set of window attributes determine whether the window has a close box, a title bar, a zoom box, and or a grow box.

The `CloseBox` attribute determines whether the window has a close box. Regular and floating windows typically do. Modal windows typically do not.

The `TitleBar` attribute determines whether the window has a title bar. Typically, a non-moveable modal dialog does not have a title bar. All other windows have a title bar.

The `Resizable` attribute determines whether the user can manually resize the window by clicking and dragging in the bottom right

corner of the window. Note that being resizable does not automatically give the window a size box. You must set the `SizeBox` attribute to have a size box appear in the window.

The `SizeBox` attribute determines whether the window has a size box. If this attribute is set, the `Resizable` attribute should also be set. A resizable window may not have a size box, but all windows with a size box should be resizable.

The `Zoomable` attribute determines two things: whether the windows is zoomable, and if it has a zoom box. All zoomable windows have a zoom box. Modal windows are typically not zoomable. Regular and floating windows may or may not be zoomable.

Drawing attributes

Three attributes control various facets of how a window draws.

The `ShowNew` attribute determines whether a new window is initially visible or not.

The `HideOnSuspend` attribute is typically true for floating windows. When an application is suspended—sent to the background—the human interface guidelines dictate that all floating palettes should be hidden. This attribute is typically false for regular and modal windows. Regular and modal windows become inactive, but do not hide.

The `EraseOnUpdate` attribute determines how window drawing occurs. If this attribute is true, PowerPlant erases the contents of the window before drawing. This attribute is typically true for windows.

Clicking attributes

The remaining window attributes control how the window responds to clicks.

The `Enabled` attribute determines whether the window is enabled—that is, can it respond to clicks.

The `Targetable` attribute determines whether the window is or can contain a command target object—that is, can it respond to menu commands and keystrokes. Most windows are targetable. This feature isn't really enforced. For example, assume you have a window with an LTextEditView object. If this attribute is not set for the window, the LTextEditView object can still become the target object. However, when the window is deactivated and reactivated, the LTextEditView will not be restored as the target object.

The `GetSelectClick` attribute determines what you do with the click that activates the window. For regular windows, usually the click activates the window, and does nothing else. For tool palettes in the floating layer, you typically set this attribute to true. In that case, the click that activates the window is also treated as a click in the window contents (for example, a click on a button in the tool palette). This maintains the illusion that the palette is always active.

If the `DelaySelect` attribute is true, a click in an inactive window is first treated as a click in the contents. After the click event is processed, if the mouse button has been released in the window, the window is selected. You set this attribute to true to support drag and drop. In the drag and drop human interface, you should be able to drag selected information out of a background window without activating the window.

WARNING! To support dragging from an inactive window, you must override LPane::Click(). See the PowerPlant Advanced Topics chapter on Drag and Drop for a thorough discussion of this issue.

Setting window attributes

The LWindow class has three accessors related to attributes. You use these accessors to determine the state of the window's attributes, or to modify the attributes at runtime. The functions and their purpose are listed in Table 11.2.

Table 11.2 Window attribute accessors

Function	Purpose
`HasAttribute()`	returns whether the attribute is set
`SetAttribute()`	set the specified attribute
`ClearAttribute()`	clear the specified attribute

Typically you set the attributes in Constructor when you design the visual interface for your application, or before creating a window on the fly. Setting or clearing attributes does not have an immediate effect on a window that has already been created. For example, setting the `windAttr_CloseBox` attribute doesn't suddenly give the window a close box.

Window Size and Zooming

In addition to the many window attributes, a window has its own unique way of keeping track of its dimensions.

Windows may have a minimum and maximum size (although this feature is only important for resizable windows). Windows also have a standard size—used for zooming.

Minimum and maximum sizes

These sizes control how large or small the window may become as the user resizes the window manually. If the window is not resizable, these values are not used.

The minimum and maximum sizes are stored in a single Rect data member named `mMinMaxRect`. The top and left fields of the Rect define the minimum size. The bottom and right fields of the Rect define the maximum size. For example, if the `mMinMaxRect` fields were {100,100,300,400}, then the minimum window size would be 100 x 100 pixels, and the maximum window size would be 300 x 400 pixels.

The accessors for this data are `GetMinMaxSize()` and `SetMinMaxSize()`. Typically you set this information in Constructor when defining the window characteristics.

The minimum size should be large enough to show some meaningful content. In general, a window shouldn't be smaller than a size of 100 by 100 pixels.

The maximum size should be large enough to show all the data in the window. By default, PowerPlant sets the maximum size to 32K by 32K. In Constructor you see this value expressed as -1.

Standard size and zooming

Clicking the zoom box of a zoomable window toggles the window between its **standard state** and its **user state**. The standard state is the optimal size and position for the window. The user state is the runtime size and position of the window that the user sets manually on the desktop. The user state changes when the user drags or resizes the window.

When the user clicks the zoom box, PowerPlant checks whether the window is in the standard state. If it is, PowerPlant resizes the window to the user state. If the window is not in the standard state, PowerPlant resizes the window to the standard state.

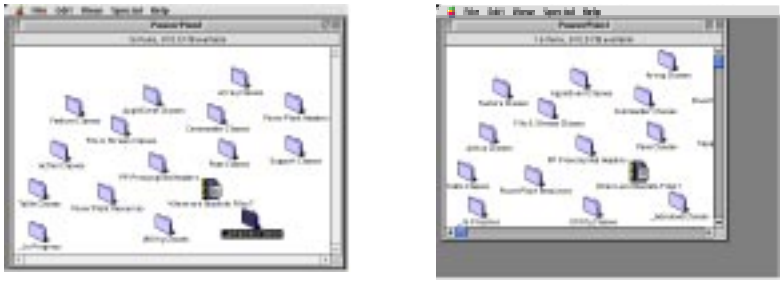
The standard size characteristic of a window defines the size of the window in its standard state. This value is stored in an `SDimension16` data member named `mStandardSize`. The accessors are `GetStandardSize()` and `SetStandardSize()`.

Typically you set this information in Constructor when defining the window characteristics. The default standard size is 32K by 32K. In Constructor you see this value expressed as -1.

When PowerPlant resizes a window to its standard state, PowerPlant uses either the standard state value or the size of the dominant screen, whichever is smaller. The dominant screen is the screen that contains most of the window. If the screen dimensions are used, PowerPlant allows a two-pixel margin around the edges of the screen and the menu bar.

For example, if the standard state of a window is 600 by 400 pixels, it will fit comfortably on a 13-inch monitor. If most of the window is on a Classic-sized monitor, PowerPlant resizes the window to 508 by 318 pixels.

Figure 11.2 Standard and User States



Standard State User State

Window Descriptor

As you know, every pane has `GetDescriptor()` and `SetDescriptor()` accessors for the pane's descriptor characteristic. The LWindow class uses these accessors to retrieve or modify the window title.

Once again, you can set the title in Constructor. However, this is one case where you are likely to use the accessors fairly regularly in your code. The title of a document-related window is likely to change as the user saves the document to a file.

Window Kind

To help you distinguish PowerPlant windows from non-PowerPlant Macintosh windows, PowerPlant stores a special value in the WindowRecord's `windowKind` field. PowerPlant assumes that any window whose `windowKind` is greater than or equal to `PP_Window_Kind` is a PowerPlant window. This constant is defined in `LWindow.h`, and its value is 20000.

You can use this feature to keep track of different kinds of PowerPlant windows in your application, if that is necessary in your de-

sign although you could accomplish the same goal using the pane ID. Using the window kind is the only way to distinguish subclasses of LWindow at runtime using a WindowRecord. Just give different kinds of windows unique window kinds greater than `PP_Window_Kind`. You can also use this feature to distinguish between PowerPlant and other windows. We'll discuss how to set the `windowKind` field in "Creating a Window" on page 362.

To get a PowerPlant window's window kind, use LWindow's `GetMacPort()` member function. Cast the returned `GrafPtr` to a `WindowPeek` so you can access the `windowKind` field.

Listing 11.1 Getting the window kind

```
WindowPeek thePeek = (WindowPeek)theWindow-> GetMacPort();
short theKind = thePeek->windowKind;
```

Working With Windows

Because you already know about how to work with panes, views, and commanders, you already know most of what there is to know about windows. However, LWindow does have its own behaviors in addition to those inherited from other classes.

In this section we discuss:

- Creating a Window—using Constructor, creating a window on the fly, and deriving window classes.
- Drawing a Window and Its Contents—things to consider when drawing a window, including offscreen drawing.
- Managing Window Behavior—doing all those things that windows do.
- Window Utilities in PowerPlant—the functions of the UWindows and UDesktop classes.
- Dealing with the Window Manager—how to get at the Mac OS window-related structures you may need on occasion.

Creating a Window

You can create a window using Constructor, or on the fly in your code. We talk about each method. Then we discuss what you do when you derive your own class from LWindow.

Using Constructor

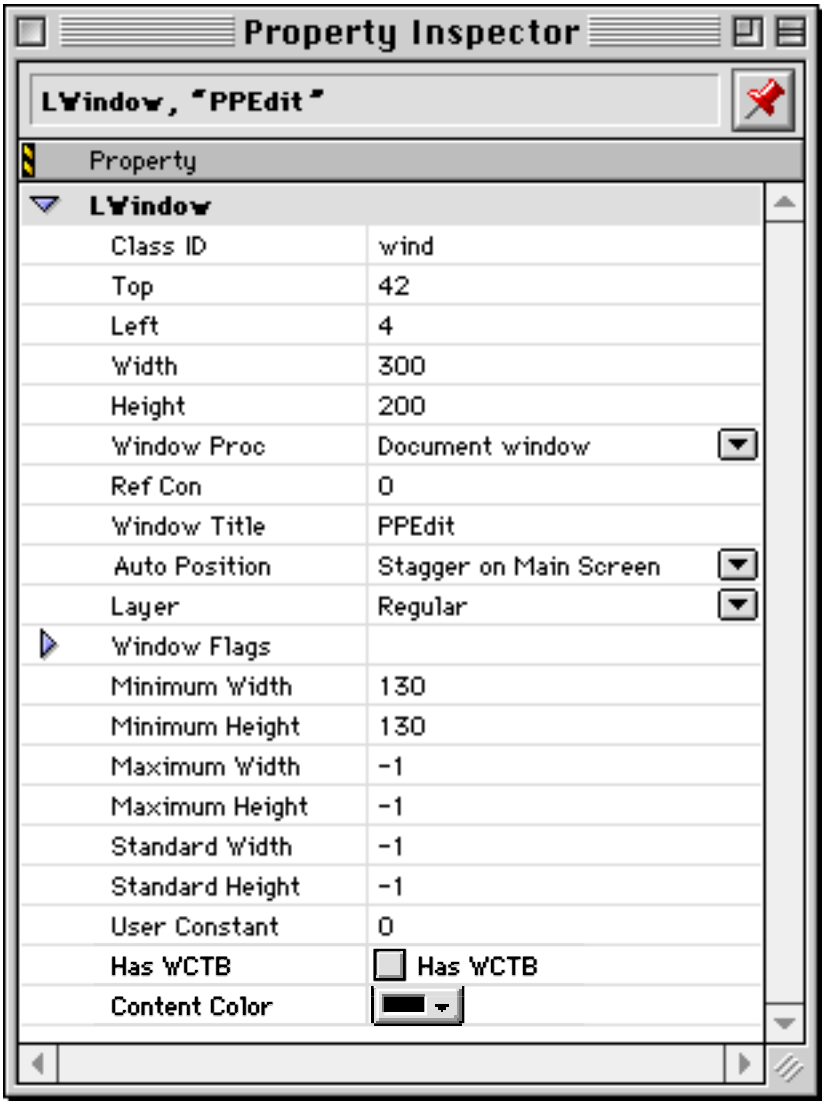
Creating a window in Constructor is simple.

While in the Constructor project window, select the Windows and Views resources, and choose **New Resource** (command-K) from the **Edit** menu. Fill in the details in the Create New Resource dialog to create a new PPob resource.

Figure 11.3 Creating a new window

Double-click the new PPob resource to see the layout editor for the new window. Double-click the new window in the layout editor to see and set the window characteristics, as shown in Figure 11.4.

Figure 11.4 Setting window properties with Constructor

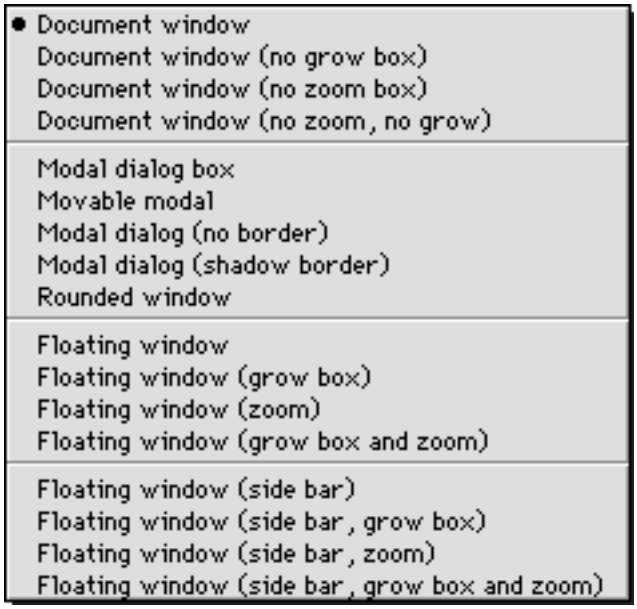


Note that a window has some of the same information as a pane, including location, size, and class ID. There is no binding information, because this is the top view. There is no pane ID for the same reason. Remember, when you derive your own classes you must change the class ID to your own unique value and register the class with PowerPlant before creating any objects of that class.

All of the window attributes we discussed earlier are available. You select a window layer. You select the peripheral window parts for your window. You set the other clicking and drawing attributes. You set the minimum, maximum, and standard sizes. And you set the window title.

You can set the window proc by choosing an item from the popup menu illustrated in Figure 11.5.

Figure 11.5 Window kind options


Document window
Document window (no grow box)
Document window (no zoom box)
Document window (no zoom, no grow)
Modal dialog box
Movable modal
Modal dialog (no border)
Modal dialog (shadow border)
Rounded window
Floating window
Floating window (grow box)
Floating window (zoom)
Floating window (grow box and zoom)
Floating window (side bar)
Floating window (side bar, grow box)
Floating window (side bar, zoom)
Floating window (side bar, grow box and zoom)

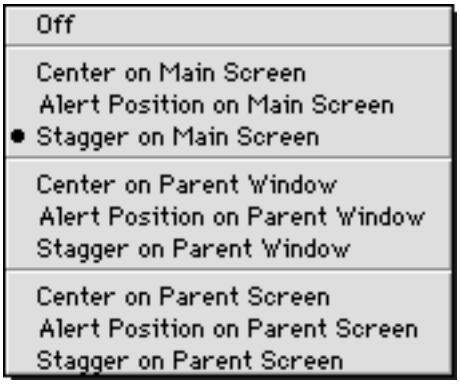
For regular windows, you choose the Document window item. The movable modal type is most appropriate when creating dialog boxes. Making a floating window is as simple as choosing the correct window type and putting the window in the floating layer. The "side bar" type puts the floating window's "title bar" on the side of the window rather than across the top.

Use the floating window types exclusively for floating windows. Good interface design dictates that floating windows look different from other windows, because they behave differently.

NOTE: To use a floating window, you must also use the UFloatingDesktop.cp implementation of the UDesktop class. See "UDesktop" on page 375.

You can use System 7's auto-positioning feature for windows. Choose an item from the popup menu illustrated in Figure 11.6.

Figure 11.6 Window positioning options


Off
Center on Main Screen
Alert Position on Main Screen
Stagger on Main Screen
Center on Parent Window
Alert Position on Parent Window
Stagger on Parent Window
Center on Parent Screen
Alert Position on Parent Screen
Stagger on Parent Screen

Finally, you can set initial values for a user constant or the window `refCon`. The latter represents the classic Mac OS WindowRecord `refCon` field. PowerPlant uses the `refCon` field to store a pointer to the LWindow object. You cannot use the `refCon` for other purposes. Use the `userCon` field instead.

Use the `refCon` field in Constructor if you want your window to have a unique window kind characteristic. Make sure the value is greater than `PP_Window_Kind`. When PowerPlant creates the window object, it moves the value from the `refCon` field of the resource to the `windowKind` field in the WindowRecord, and replaces the `refCon` with a pointer to the window object.

All of the possibilities for all of the characteristics give you a plethora of possible window designs. Creating unique and interesting windows is as simple as making a few choices in the pane Property Inspector window for the LWindow object.

With a PPob resource, creating the window (and all of its contents) in code is simple. Call LWindow's class creator function, `LWindow::CreateWindow()`. PowerPlant does the rest.

See also "Register PowerPlant Classes" on page 288.

Creating a window on the fly

If you wish to create a window on the fly, you use the `SWindowInfo` structure, detailed in Listing 11.2. This structure specifies the values required to build a window. You define an `SWindowInfo` structure, and fill in the values. In the structure you provide the resource ID of a WIND resource, the layer in which to place the window, the attributes, the minimum, maximum, and standard sizes, and the `userCon` value.

To set the window kind from a resource, set the `refCon` value in the window's WIND resource to be your desired window kind. You can also change the `windowKind` field directly after creating the window. Make sure the value is greater than `PP_Window_Kind` for a PowerPlant window.

Then you call the LWindow constructor. There is a constructor that takes a reference to an `SWindowInfo` structure as a parameter.

Listing 11.2 The SWindowInfo structure

```
struct SWindowInfo {
  ResIDT       WINDid;
  Int16        layer;
  Uint32       attributes;
  Int16        minimumWidth;
  Int16        minimumHeight;
  Int16        maximumWidth;
  Int16        maximumHeight;
  SDimension16 standardSize;
  Int32        userCon;
};
```

You can use LWindow-specific calls to set or modify other information about the window after you create it. After you have built the window, you can add whatever panes and views you wish.

Deriving your own windows

Creating your own window class is a fairly common occurrence. There are many reasons why you might want to extend the functionality of the basic LWindow class.

For example, you might want a window that can listen to control items (like an LDialogBox object). You might want your window to zoom or grow in unusual ways. There's no telling what you might want to do with a window.

In the case of deriving from LWindow, you are very likely to override the functions listed in Table 11.3.

Table 11.3 Commonly overridden LWindow functions

Function	Purpose
`ClickSelf()`	interaction outside of subpanes
`FinishCreateSelf()`	complete window setup
`FindCommandStatus()`	handle menu updating
`ObeyCommand()`	respond to window-level commands

You are familiar with all four of these functions from our earlier discussions of panes, views, and commanders.

Beyond the functions listed in Table 11.3, it is difficult to typify the other functions you will override. Suffice it to say, if your window has unique behavior, you override those functions necessary to implement that behavior. To determine what functions you need to override, explore the LWindow class to see how the default behavior is implemented. We discuss some of these functions in the "Managing Window Behavior" on page 370.

For example, if you want to modify how a window resizes (perhaps to allow switching between a few specific sizes) you should examine `LWindow::ClickInGrow()`, `LWindow::DoSetBounds()`, and `LWindow::DoSetZoom()` to see if you should override them.

Drawing a Window and Its Contents

A window is a view. Drawing a window is just like drawing a view. You call the window's `Draw()` method. LWindow does not override this method. The inherited `LView::Draw()` function first calls the window's own `DrawSelf()` function, then walks through the list of subpanes and tells each of them to `Draw()`.

The `LWindow::DrawSelf()` function erases the window contents (if the `EraseOnUpdate` attribute is set), and then draws the size box (if there is one). The net effect is the window erases its contents, and then the subpanes draw themselves into the empty window.

Offscreen Drawing

In certain situation, you may want to use LOffscreenView for drawing. You can put an LOffscreenView object into the window to enclose several other panes. When you draw the subpanes, they all draw in the offscreen view. When all panes are finished drawing, the complete image is blitted to the screen. Even though this is a trifle slower than drawing the individual panes directly on screen, the net effect appears faster to the user because all the panes appear simultaneously. This technique is also useful if the panes overlap and might cause flicker while drawing.

LOffscreenView creates and destroys a temporary GWorld every time you draw. If you want a view or panes within the view to use a GWorld that lasts for more than one update, use LGWorld.

Examine the function definitions in the `UGWorld.cp` file. For example, if you wanted an individual pane to use an LGWorld, you would create a new LGWorld object from the constructor, and delete the LGWorld object in the destructor. You could then use the LGWorld to store the visual image of the pane. Listing 11.3 is one example of how you might do this.

Listing 11.3 Simple example for LGWorld

```
MyPane::MyPane() {
  Rect frame;
  CalcLocalFrameRect(frame);
  mGWorld = new LGWorld(frame, 8);
}

MyPane::~MyPane() {
  delete mGWorld;
}

MyPane::DrawSelf() {
  Rect frame;
  CalcLocalFrameRect(frame);
  mGWorld->CopyImage(GetMacPort(), frame);
}
```

This hypothetical MyPane class would need a new function to draw the contents of the pane into the GWorld in the first place, and to update those contents when necessary. Such a function might look something like Listing 11.4.

Listing 11.4 Drawing in an LGWorld object

```
MyPane::DrawInGWorld {
  mGWorld->BeginDrawing(); // Draw in GWorld
  // code to draw pane
  mGWorld->EndDrawing();
```

You use the LGWorld function `BeginDrawing()` to prepare the GWorld, and `EndDrawing()` when you're finished. All the drawing in between occurs in the GWorld and not on screen.

TIP: LGWorld is another independent PowerPlant class. You can use LGWorld without using any other part of PowerPlant.

See also "LOffscreenView" on page 195.

Managing Window Behavior

Windows have several common behaviors with which you are familiar, including activating, dragging, resizing, zooming, and closing. In this section we examine the behavior of an LWindow object and the default implementation of these functions in LWindow. Unless your window does something unusual you shouldn't have to override this behavior.

Selecting, showing, and hiding a window

Remember the three facets of a pane's state: visible/hidden, active/inactive, enabled/disabled. All of these apply to windows. Windows, of course, do unique things to manage state, but the general principles remain the same.

In addition, you may select a window, and respond when an application as a whole suspends or resumes. Table 11.4 lists some of the functions related to the window's state.

Table 11.4 Some LWindow state-related functions

Function	Purpose
`Select()`	bring window to front
`Suspend()`	behavior when application suspends
`Resume()`	behavior when application resumes
`ShowSelf()`	make window visible
`HideSelf()`	make window invisible
`ActivateSelf()`	make window active
`DeactivateSelf()`	make window inactive

There are two points to make about these functions.

First, the four "self" functions are protected member functions. Typically you would never call them directly. They are listed here for information only. You should use the corresponding public inter-

face of `Show()`, `Hide()`, `Activate()`, and `Deactivate()`, each of which calls the corresponding "self" routine. To implement unique behavior in a derived class, you would override the "self" routines.

Second, if you examine the source code for these functions, you'll see that several of them call static members of the UDesktop class. UDesktop encapsulates much of PowerPlant's window management behavior.

See also the discussion of "State" on page 147 and "UDesktop" on page 375.

Handling clicks

A click in a window is, in many ways, the defining event in an application. What you do in response to a click determines how your application behaves.

Clicks in a PowerPlant window can be thought of as occurring in one of two general locations: in the peripheral window parts or in the window content.

LWindow has default behaviors for handling most if not all of these situations for you. `LWindow::HandleClick()` parses the click and dispatches control based upon the location of the click.

Click in a peripheral control

A click in a peripheral control ultimately results in one of these functions being called, whichever is appropriate for the click:

- `LWindow::ClickInDrag()`
- `LWindow::ClickInZoom()`
- `LWindow::ClickInGrow()`
- `LWindow::ClickInGoAway()`

If you examine the code for these functions, you'll see that each of them uses Apple events while implementing their respective behavior. As a result, these actions are scriptable and recordable in a PowerPlant application.

The default implementation of these functions in LWindow is likely to suit your needs just fine. If it does not, you can derive your own window class and override whichever behavior does not fit your precise needs.

See also "Closing a window" on page 372.

Click in the content area of a window

`LWindow::ClickInContent()` handles a click in the content area of a window. This function selects the window if necessary. It then calls `Click()` to handle the click. LWindow does not override `Click()`, so it uses `LView::Click()`.

`LView::Click()` identifies whether the click occurred inside a pane within the window. If it is within a pane, the function calls the pane's `Click()` function so the pane can respond.

If the click is within the window contents but not inside a pane, control passes one step higher up the inheritance chain to `LPane::Click()`. After performing some housekeeping details, `LPane::Click()` calls `ClickSelf()`.

Neither LWindow nor LView overrides `ClickSelf()`. Therefore, the `LPane::ClickSelf()` function executes. The default implementation of `LPane::ClickSelf()` does nothing. If you want your window to respond to clicks outside of any subpanes, override `ClickSelf()`.

Closing a window

LWindow has two functions associated with closing a window, `AttemptClose()` and `DoClose()`. These very similar functions are used at different times and in different circumstances. Understanding the differences can help you decide which to use, and which to override.

Listing 11.5 displays the code for `AttemptClose()`.

Listing 11.5 LWindow::AttemptClose() function

```
if ((mSuperCommander == nil) ||
      mSuperCommander->AllowSubRemoval(this))
{
  // Send Close AE for recording only
  SendSelfAE(kAECoreSuite, kAEClose, false);
  delete this;
}
```

The code for `DoClose()` is almost identical.

Listing 11.6 LWindow::DoClose() function

```
if ((mSuperCommander == nil) ||
      mSuperCommander->AllowSubRemoval(this))
{
     delete this;
}
```

The difference between these functions is that `AttemptClose()` sends an Apple event. As a result, a scripting environment can record the window closure. Otherwise, the functions are identical.

Within PowerPlant, `AttemptClose()` is called in response to a click in the close box of a window. This reflects PowerPlant's support for script recordability.

The `DoClose()` function is called in two cases: when `LDialogBox::ListenToMessage()` receives a close message; and when the window receives an Apple event to close.

If you want to close a window, call `AttemptClose()`. This is your best bet to ensure your application's recordability. If you override window-closing behavior, `AttemptClose()` is again the best bottleneck. However, good design dictates that the same perceived behavior should occur whether the user clicks in the close box, or the window receives a close Apple event. To ensure that the same behavior occurs, you may need to override `DoClose()` as well, depending upon what you do in your override of `AttemptClose()`.

Finally, note that each function calls the commander's `AllowSubRemoval()` function. Why? For example, in a typical implementation a document is the supercommander of a window. This gives the document an opportunity to check whether there are any changes, and gives the user the chance to save them.

Because *both* `AttemptClose()` and `DoClose()` pass through the `AllowSubRemoval()` bottleneck, you can institute your required closing behavior—such as a "save changes" check—in this function once, and leave `AttemptClose()` and `DoClose()` alone.

WARNING! If the floating window is targetable, or it contains the current target object (such as an editable text field), when you issue a Close command it will operate on the floating window! That's because this is the window that contains the target object. You may want to override the default closing behavior to close the frontmost regular window in this case. Your user interface will determine precisely how you should handle the close behavior.

Window Utilities in PowerPlant

PowerPlant has two classes that contain a variety of utilities for managing window-related tasks. They are UWindows and UDesktop. All of the functions in both of these classes are static, so you can call them at any time.

UWindows

The UWindows utility functions relate to Mac OS WindowRecords, not to PowerPlant LWindow objects. You can use the UWindows functions without using any other part of PowerPlant.

The functions in UWindows relate to finding the dimensions or front-to-back order of an individual Mac OS window record. There is also a function for finding the device on which most of a specified rectangle appears. Table 11.5 lists the functions.

Table 11.5 UWindows functions

Function	Purpose
`GetWindowContentRect()`	return the bounding rectangle of the content region
`GetWindowStructureRect()`	return the bounding rectangle of the structure region
`FindDominantDevice()`	return the GDevice which contains the largest portion of the specified rectangle
`FindNthWindow()`	return a WindowPtr to the nth window
`FindWindowIndex()`	return index position of a window
`FindNamedWindow()`	return a WindowPtr to the window with the specified name

Consult *PowerPlant Revealed* for details on how to use these functions.

UDesktop

UDesktop encapsulates much of PowerPlant's low-level window management behavior. Unlike UWindows, UDesktop is dependent upon the LWindow class. Where functions in UWindows typically have a Mac OS WindowPtr as a parameter or return value, UDesktop uses pointers to LWindow objects. As a result, you can't use UDesktop without using a significant part of PowerPlant.

Just like UDesktop relies on LWindow, LWindow relies on the existence of the UDesktop functions. You *must* include some file in your project which implements the class defined in UDesktop.h. PowerPlant has two versions of UDesktop. You can use UDesktop.cp *or* UFloatingDesktop.cp, but not both. If you attempt to include both, you will get a link error for multiple definitions of the same class. You can substitute your own implementation of UDesktop if you wish.

TIP: The implementation of UDesktop found in UDesktop.cp does not support floating windows. Use UFloatingDesktop.cp in your project if you use floating windows.

Table 11.6 lists all the member functions of UDesktop. Remember that in this context the term "window" refers to an LWindow object, not a Mac OS WindowRecord, unless otherwise specified.

Table 11.6 UDesktop functions

Function	Purpose
`NewDeskWindow()`	create a new Mac OS window
`WindowIsSelected()`	returns whether window is at the top of its layer
`SelectDeskWindow()`	bring window to top of its layer and activate it
`ShowDeskWindow()`	make a window visible
`HideDeskWindow()`	make a window invisible
`DragDeskWindow()`	drag a window
`Suspend()`	suspend all windows
`Resume()`	resume all windows
`Deactivate()`	deactivate all windows
`Activate()`	reactivate appropriate windows
`FetchTopRegular()`	return top regular window
`FetchTopFloater()`	return top floating window
`FetchBottomFloater()`	return bottom floating window
`FetchTopModal()`	return top modal window
`FetchBottomModal()`	return bottom modal window

Function	Purpose
`FrontWindowIsModal()`	returns whether front window is modal
`NormalizeWindowOrder()`	restore window order

You are free to call UDesktop functions directly at any time if they suit your purpose. Most of the time, you won't need to use them directly. PowerPlant uses these as utility functions to perform what are, for the most part, low-level window management tasks. However, you might find some of them useful in particular circumstances where you need to know some desktop detail, such as whether the front window is modal or not.

Dealing with the Window Manager

When dealing with windows in a PowerPlant application, you must still work directly with the Mac OS Window Manager from time to time. PowerPlant provides several functions for dealing with the Mac OS GrafPort, or an LWindow object's associated WindowPtr.

Table 11.7 lists the available functions.

Table 11.7 GrafPort and WindowPtr utility functions

Function	Purpose
`UQDGlobals::GetCurrentPort()`	return current GrafPort from QuickDraw globals
`LPane::GetMacPort()`	return WindowPtr containing this pane
`LWindow::GetMacPort()`	return WindowPtr associated with this LWindow object
`LWindow::FetchWindowObject()`	return a pointer to the LWindow object associated with the specified WindowPtr

Both the `GetCurrentPort()` and `FetchWindowObject()` functions are static functions, so they are always available.

Using these four functions in appropriate combination you can do some interesting things.

For example, you can always find the front window object by getting the current port and then finding the associated window object. If you have a pane object, you can get the Mac OS WindowPtr for the window that contains this pane by calling `GetMacPort()`.

You may recall a discussion about "Finding the topmost view" on page 188. If the topmost view is a window, you can get the topmost view this way:

Listing 11.7 Finding the containing LWindow object

```
GrafPtr theWindowP = thePane->GetMacPort();
LWindow* topView = LWindow::FetchWindowObject(theWindowP);
```

PowerPlant is a flexible tool. In many cases PowerPlant gives you several ways to solve a problem. Getting the topmost window object is but one example.

Summary

In this chapter you learned all about windows in PowerPlant. We discussed what a window is, and the LWindow class hierarchy.

We discussed the additional features of a window that distinguish it from other views or panes, including special window attributes, special sizes, and the LWindow use of the descriptor characteristic.

Window attributes control a window's layer, the controls around the edges of the window, how the window draws, and how it handles clicks.

We discussed how to work with windows, including how to create windows, draw windows, draw off screen, handle clicks, and close a window.

Finally, we discussed a variety of window-related utility functions in PowerPlant that allow you to manage window behavior, and work with the Mac OS Window Manager.

Code Exercise

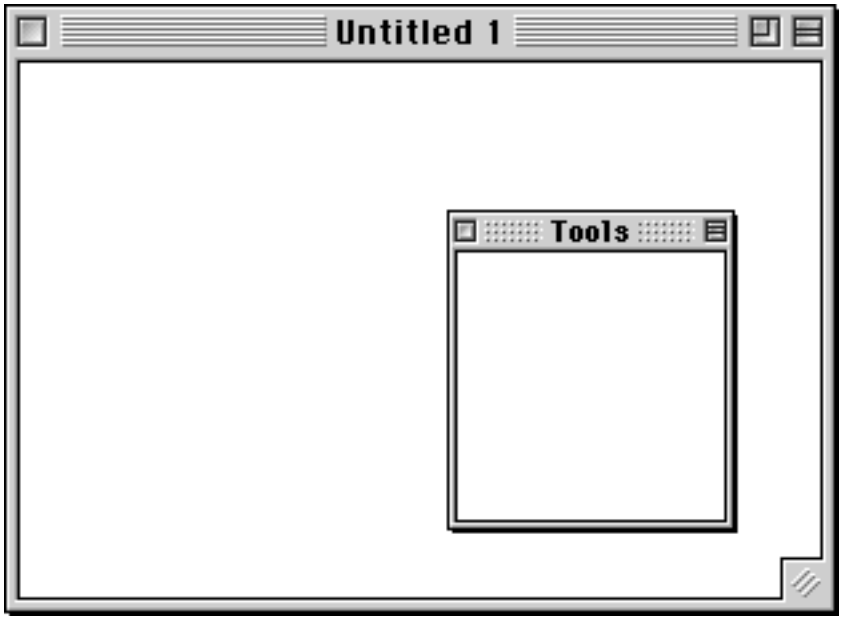
In this code exercise you write an application named "Windows." In the process you implement a feature that is common to many applications—a **Window** menu that lists open windows. You also write the code to create both a floating window and a regular window.

As you have in several other exercises, you'll accomplish this in two sections. First you examine the interface, and then you write the code.

The Interface

The final application creates two different kinds of windows, as shown in Figure 11.7: a regular window and a floating window.

Figure 11.7 The windows in "Windows"



Both windows are empty. This exercise concentrates on creating windows and working with menus.

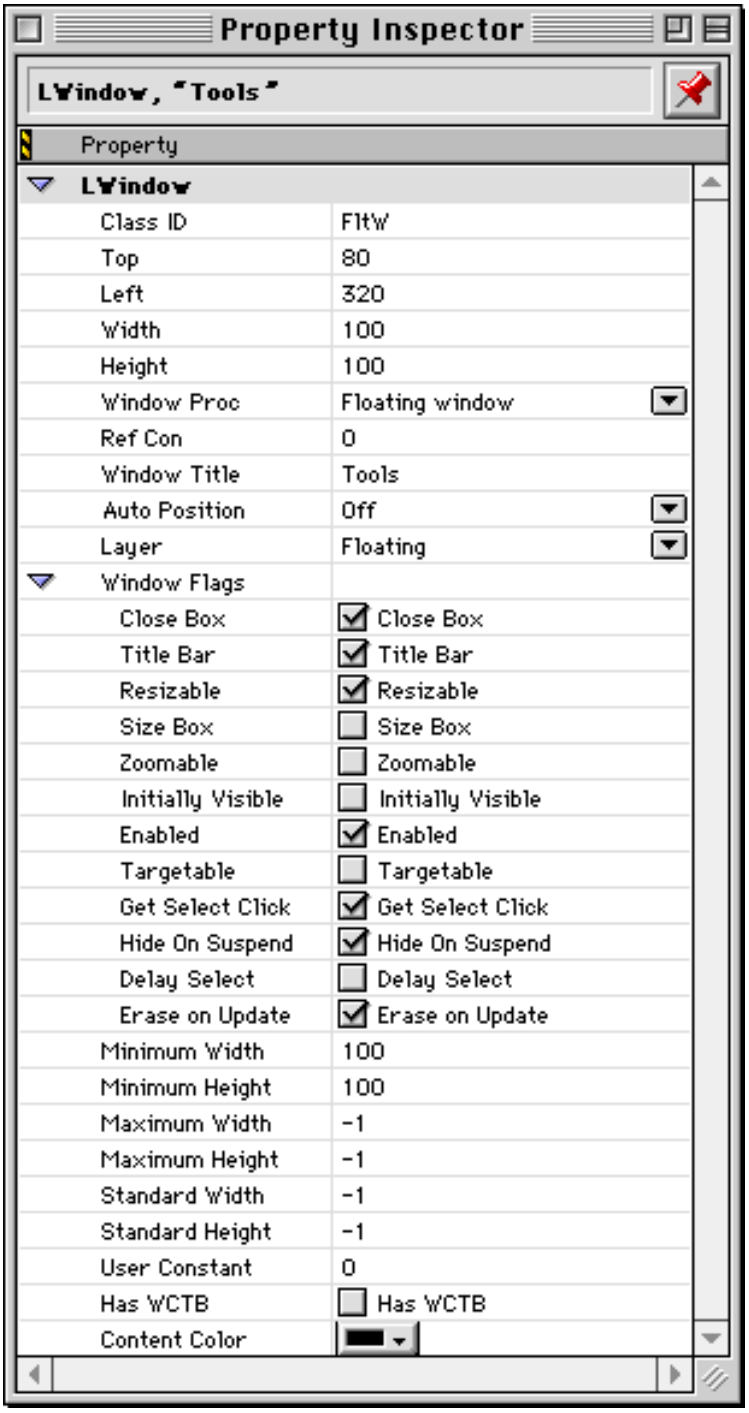
Open the `Windows.ppob` project file in Constructor and examine the two PPob resources. PPob resource ID 1000 is the regular window. Examine the characteristics of this window.

The features of this window are pretty common. The window kind is a document window, and the window is in the regular window layer. The window uses the auto-position feature. In the clicking and drawing section, notice that the window does not get the select click and does not hide on suspend.

The window has a class ID of RegW. This is a custom window with some unique behaviors. In particular, the `FinishCreateSelf()` and destructor functions are different from LWindow. These functions add or remove the window in the **Window** menu.

Now, open the PPob for the floating window, and examine the window's characteristics as illustrated in Figure 11.8.

Figure 11.8 The floating window properties



There are several differences between the Tools window and the regular window. Most importantly, the window kind is a floating window and it is in the floating layer. These two features control most of the window's basic appearance and behavior. However, there are other important characteristics.

The window is not targetable or initially visible. The window has a close box and a title bar. There is no size box or zoom box. The window is not resizable. A typical floating window does not resize or zoom.

Next, observe the clicking and drawing features. This window gets the select click. As a result, a click that activates the window is also treated as a click in the window content. You'll see how this works when you write code to respond to a click.

This window also hides when the application is suspended. Again, this is not mandatory. This feature reflects the typical behavior of a floating window in the Mac human interface.

Finally, this is a custom window with class ID FltW. In this case, the difference between this window and the behavior of LWindow is in the `ClickSelf()` and `ClickInGoAway()` functions. You'll write the code for these functions a little later in this exercise.

You can close the PPob windows and the Constructor project file.

The Windows Application

Before we get started on the actual code, let's take a quick look at where we're going so you know where all the pieces fit.

The **Window** menu is implemented as a custom class derived from LMenu. CWindowMenu has some additional features we'll explore in just a bit. The application can't use the PowerPlant default menu-creation mechanism because PowerPlant creates LMenu objects. This application creates a CWindowMenu object explicitly and adds it to the menu bar. The application stores a pointer to the CWindowMenu object in a global variable, `gWindowMenu`.

When the user creates or destroys a regular window, the window title is added or removed as an item in the **Window** menu. You write the code to make that happen. You also write the code that allows the application to run everything. Let's get to it.

1. **Examine CWindowMenu.**

`class declaration` `CWindowMenu.h`

When you look at the class declaration, you see that this class inherits from LMenu. In addition to the usual constructors and destructor, CWindowMenu declares five new functions. They are:

- `InsertWindow()`
- `RemoveWindow()`
- `MenuItemToWindow()`
- `WindowToMenuItem()`
- `SetCommandKeys()`

You write `InsertWindow()` and `RemoveWindow()` in the next two steps to add or remove items from the **Window** menu. The other three functions are provided for you. `MenuItemToWindow()` returns the LWindow pointer for a given menu item. `WindowToMenuItem()` returns the correct menu item for a given LWindow pointer. `SetCommandKeys()` assigns numerical command keys to the first nine open windows.

There are two new data members as well, `mBaseItems` and `mWindowList`.

The first item in the **Window** menu is an item to show or hide the Tools window. If there is a regular window open, the next item in the **Window** menu is a separator bar. These are the "base" items tracked in the `mBaseItems` member. All other items in the menu match the titles of various open windows.

The `mWindowList` member is a list of open regular windows.

When you are through studying, close the file.

2. Add a window to the menu.

`InsertWindow()` `CWindowMenu.cp`

The existing code in this function first determines whether the window is already in the list. If it is not, you must accomplish these tasks.

a. Add the window to the window list.

Use `mWindowList`, and call `InsertItemsAt()` to add a new item to the list.

b. Get the window title.

Use the window's `GetDescriptor()` function.

c. Add an item to the Window menu.

This is a bit more complex. You use the CWindowMenu's inherited `InsertCommand()` function. However, it calls the Toolbox `InsertMenuItem()` function. The Toolbox function recognizes and uses "metacharacters" so you can accomplish tasks like setting a mark or command key when you insert an item. Unfortunately, one or more characters in your window title might be misinterpreted as a metacharacter, causing odd results.

To avoid this problem, when you call `InsertCommand()`, pass a blank space as the text for the new menu item. You also specify a menu command. Use `cmd_UseMenuItem`. Add the item to the end of the menu.

After the call to `InsertCommand()`, use the Toolbox `SetMenuItemText()` function to change the text for the item to match the window title.

d. Adjust the command keys.

The **Window** menu has a feature that assigns numerical command keys to the first nine windows in the window list. Call `SetCommandKeys()` to implement this behavior.

Existing code handles the `else` condition when the window is already in the list. It changes the text of the menu item to match the window title. As a result, you can use this function for two purposes: to add a new window to the menu, or to modify the menu when the window title changes.

```
      mBaseItems++;
    }
  }

  // Add the window to the list.
  mWindowList.InsertItemsAt( 1, arrayIndex_Last, &inWindow );

  // Get the window title.
  Str255 theTitle;
  inWindow->GetDescriptor( theTitle );

  // Insert title into the menu as a -1 item.
  InsertCommand("\p ", cmd_UseMenuItem, 16000 );
  ::SetMenuItemText( GetMacMenuH(),
            ::CountMItems( GetMacMenuH() ), theTitle );

  // Renumber the command keys.
  SetCommandKeys();

} else { // Already in list
```

3. **Remove a window from the menu.**

`RemoveWindow()` `CWindowMenu.cp`

Existing code ensures that the window you are removing actually exists. If it does, you have two tasks to accomplish.

a. Remove the menu item corresponding to the window.

The order of window titles in the menu matches the order of windows in the window list. Get the index number of the window from the window list. Use the menu object's inherited `RemoveItem()` function to remove the item from the menu. Don't forget to add `mBaseItems` to the index value of the window to get the correct menu item number.

b. Remove the window from the window list.

Use `mWindowList`, and call `Remove()` to remove the specified window from the list.

```
// Remove the item from the menu.
Assert_( mWindowList.FetchIndexOf(
                &inWindow ) != arrayIndex_Bad );
RemoveItem( mWindowList.FetchIndexOf(
              &inWindow ) + mBaseItems );

// Remove the window from the list.
mWindowList.Remove( &inWindow );

if ( mWindowList.GetCount() == 0 && mBaseItems > 1 ) {
```

The existing code then handles the separator, removing it when there are no more windows in the **Window** menu. The existing code also resets the command keys for the open windows.

When you are through, save your work and close the file. It's time to implement window behavior.

4. **When creating a window, add it to the menu.**

`FinishCreateSelf()` `CRegularWindow.cp`

When the user creates a regular window, you want to add it to the **Window** menu. The `FinishCreateSelf()` function is designed for the tasks you must accomplish to finish a pane, view, or control.

The Mac human interface says that new windows not associated with any file on disk should be named "Untitled" followed by a number. The existing code uses the `UWindows::FindNamedWindow()` utility to ensure that you have a unique and appropriate title to assign to your window.

When you have a good title, you have two tasks to accomplish.

a. Set the window title.

Use the window object's `SetDescriptor()` function.

b. Add the window to the Window menu.

Use the `gWindowMenu` global variable. (You'll initialize this variable in a subsequent step.) Use the CWindowMenu object's `InsertWindow()` function.

```
  theTitle += (LStr255) theNumber;
}

// Set window title.
SetDescriptor( theTitle );

// Add the window to the window menu.
gWindowMenu->InsertWindow( this );
```

5. **When destroying a window, remove it from the menu.**

   `~CRegularWindow()` `CRegularWindow.cp`

   Use the `gWindowMenu` global variable. (You'll initialize this variable in a subsequent step.) Use the CWindowMenu object's `RemoveWindow()` function.

```
CRegularWindow::~CRegularWindow()
{
  // Remove the window from the window menu.
  gWindowMenu->RemoveWindow( this );
}
```

   You have completely implemented all the new behavior of the CRegularWindow object. Save your work and close the file.

6. **Handle a click in the floating window content.**

   `ClickSelf()` `CFloatingWindow.cp`

   In a typical application the window's contents would usually handle the click. However, the Tools window is empty, and a click in the content has no real significance. The code you write in this step is for instructional purposes only, just so you can see how a floating window responds to a click.

   Make the window beep when there is a click in the contents.

```
#pragma unused ( inMouseDown )

  ::SysBeep (30);
```

7. **Handle a click in the floating window close box.**

`ClickInGoAway()` `CFloatingWindow.cp`

When the user clicks in the close box for the floating window, you could destroy the window. When the user wants to see the Tools window again, you would have to build it from scratch. This application uses a different strategy. It creates the window once, and then shows or hides the window.

Existing code calls the Toolbox `TrackGoAway()` function. If the call returns true, hide the window.

There is one other task you must perform. You must set the menu update flag as well. PowerPlant does not update menus for clicks in the title bar or peripheral controls. However, a click in the close box should modify the **Window** menu. The first item in the **Window** menu says (alternatively) either **Show Tools** or **Hide Tools**.

```
if ( ::TrackGoAway( GetMacPort(), inMacEvent.where ) ) {

  // Hide the window.
  Hide();

  // Update the menus.
  SetUpdateCommandStatus( true );
}
```

You'll create and show the window in subsequent steps. However, you have fully implemented the unique behavior of the floating window. Save your work and close the file. All that remains is to use these items—the menu, the regular window, and the floating window—in the finished application. (See "When To Update Menus" on page 325.)

8. **Install the Window menu.**

`Initialize()` `CWindowsApp.cp`

As we mentioned at the start of this section, you cannot rely on the PowerPlant menu-creation mechanism because it creates LMenu objects. The **Window** menu is a CWindowMenu object.

In the application constructor, existing code registers the custom classes. After that, in the `Initialize()` function you have three tasks to accomplish.

a. Create a CWindowMenu object.

Use the `new` operator. The declared constant for the MENU resource ID is `rMENU_Window`. Store the result in the global variable, `gWindowMenu`. It's always wise to check that creation was a success. You can use `ThrowIfNil_`.

b. Get the application's LMenuBar object.

Use `LMenuBar::GetCurrentMenuBar()`.

c. Add the new menu to the menu bar.

Use the menu bar's `InstallMenu()` function.

```
// Make the window menu.
gWindowMenu = new CWindowMenu( rMENU_Window );
ThrowIfNil_( gWindowMenu );

// Get the menu bar.
LMenuBar *theMBar = LMenuBar::GetCurrentMenuBar();
ThrowIfNil_( theMBar );

// Install the window menu.
theMBar->InstallMenu( gWindowMenu, 0 );
```

9. **Create the Tools window.**

`Initialize()` `CWindowsApp.cp`

As we discussed above, this application's strategy for the Tools palette is to create the window once, then show and hide it as neces-

sary. You should create the Tools palette in the application's `Initialize()` function.

To do so, call the `LWindow::CreateWindow()` function. The declared constant for the PPob resource is `rPPob_FloatingWindow`.

You can use the `LWindow::CreateWindow()` function because the floating window—although it has slightly different behavior—is identical to an LWindow object with respect to the data required to create the window object and its contents.

However, you should typecast the return value from `LWindow::CreateWindow()` to be a CFloatingWindow pointer. Store the result in an application data member, `mToolsWindow`.

```
theMBar->InstallMenu( gWindowMenu, 0 );

// Create the tools window.
mToolsWindow = dynamic_cast<CFloatingWindow *>
            (LWindow::CreateWindow( rPPob_FloatingWindow, this ));
ThrowIfNil_( mToolsWindow );
```

10. Create a regular window at launch.

`StartUp()` `CWindowsApp.cp`

The `StartUp()` function is called when the application launches without documents. As a result, a typical use for this function is to create a default window if the user does not open a document.

That's just what the "Windows" application does. When the application launches, a window opens automatically. To implement this behavior, simply call the application's `ObeyCommand()` function. Use the `cmd_New` command. (See "PowerPlant and Apple Events" on page 291.)

```
CWindowsApp::StartUp()
{
  ObeyCommand( cmd_New, nil );
}
```

11. Respond to commands.

`ObeyCommand()` `CWindowsApp.cp`

In this function you should respond to three commands: a selection in the **Window** menu, `cmd_New`, and `cmd_ToolsWindow`. In the substeps in this step, you handle each command.

a. Handle a Window menu command.

Recall that when you inserted an item in the **Window** menu, you assigned `cmd_UseMenuItem` as the corresponding command number. That means PowerPlant generates a synthetic command for each item.

Existing code identifies synthetic commands, and tests to ensure that the command is from the **Window** menu. If it is, you get the window object that corresponds to the menu item. Use the CWindowMenu's `MenuItemToWindow()` function. Then, if there is a window and it is visible, select the window. Use `UDesktop::SelectDeskWindow()`.

```
if (theMenuID == gWindowMenu->GetMenuID() ) {
  // Get window corresponding to the menu item.
  LWindow *theWindow =
         gWindowMenu->MenuItemToWindow( theMenuItem );

  if ( theWindow != nil && theWindow->IsVisible() ) {

    // Bring the window to the front.
    UDesktop::SelectDeskWindow( theWindow );
  }
} else { // Synthetic command not in Window menu
```

b. Create a new regular window.

The existing code has a `case` statement for `cmd_New`. In response to that command, you should create and show the window. Use the LWindow class creator function. The declared constant for the PPob resource ID is `rPPob_RegularWindow`.

```
case cmd_New:
{
  // Create the window.
```

```
  LWindow *theWindow;
  theWindow = LWindow::CreateWindow( rPPob_RegularWindow, this );
  Assert_( theWindow != nil );

  // Show the window.
  theWindow->Show();
}
break;
```

c. Show or hide the tools window.

The application receives the `cmd_ToolsWindow` command when the user selects the first item in the **Window** menu. In response to this command, you should toggle the visibility of the window. Determine if the window is visible. It it is, hide it. If it is not, show it. Use the `mToolsWindow` data member.

```
case cmd_ToolsWindow:
{
  // Toggle visible state of the tools window.
  if ( mToolsWindow->IsVisible() ) {
    mToolsWindow->Hide();

  } else {
    mToolsWindow->Show();
  }
}
break;
```

Notice that the default case for both synthetic and non-synthetic commands passes any unrecognized command to the inherited `ObeyCommand()` function for further processing.

12. Update menus.

`FindCommandStatus()` `CWindowsApp.cp`

Finally, you must update menus appropriately. You should respond to three kinds of items: **Window** menu items, the **New** item in the **File** menu, the **Tools** window item. In the substeps in this step, you handle each item.

a. Update Window menu items.

Recall that when you inserted an item in the **Window** menu, you assigned `cmd_UseMenuItem` as the corresponding command number. That means PowerPlant generates a synthetic menu command for each item. It also means that PowerPlant will call the `FindCommandStatus()` function for each item.

Existing code identifies synthetic commands, and tests to ensure that the item is from the **Window** menu. If it is, you should get the window object that corresponds to the menu item. Use CWindowMenu's `MenuItemToWindow()` function. If there is a window, enable the item, use a mark, and set the mark to `noMark`. Then, if the window is the top window, set the mark to a check mark. Use `UDesktop::FetchTopRegular()` to identify the top regular window.

```
if (theMenuID == gWindowMenu->GetMenuID() ) {

  // Find window corresponding to the menu item.
  LWindow *theWindow =
               gWindowMenu->MenuItemToWindow( theMenuItem );

  if ( theWindow != nil ) {
    // All window items enabled and use a mark.
    outEnabled = true;
    outUsesMark = true;
    outMark = noMark;

    if ( theWindow == UDesktop::FetchTopRegular() ) {

      // Check menu item for top regular window.
      outMark = checkMark;
    }
  }
} else { // Synthetic command not in Window menu
```

b. Enable the New item in the File menu.

The existing code has a `case` statement for `cmd_New`. Enable the item.

```
case cmd_New:
{
  // Enable the New command.
  outEnabled = true;
}
break;
```

c. Set the Show/Hide Tools text.

The application receives the `cmd_ToolsWindow` command for the first item in the **Window** menu. Enable the item. Set the text based on the visibility of the window. Determine if the window is visible. It it is, the item should say "Hide Tools." If the window is not visible, the item should say "Show Tools." Use the `outName` parameter to set the item text. PowerPlant does the rest.

```
case cmd_ToolsWindow:
{
  // Item is always enabled.
  outEnabled = true;

  // Toggle the menu item text.
  if ( mToolsWindow->IsVisible() ) {
    LString::CopyPStr( "\pHide Tools", outName);
  } else {
    LString::CopyPStr( "\pShow Tools", outName);
  }
}
break;
```

Notice that the default case for both synthetic and non-synthetic commands passes any unrecognized command to the inherited `FindCommandStatus()` function for further processing.

13. Build and run the application.

All right! Time to see the result of your work. Make the project and run it. When you do, an empty window should appear with the name "Untitled 1."

Look in the **Window** menu. The first item should be **Show Tools**, followed by a separator bar and the name of the regular window. Close the window and look in the **Window** menu again. How does it differ?

When you launch the application, the Tools window is not visible. Remember, the PPob for the Tools window specified that the window was not initially visible. As a result, although you create the window at launch, the window does not appear automatically.

Choose the **Show Tools** item. The Tools window appears. Look in the **Window** menu again, and the first item should say "Hide Tools."

Create a new regular window. The floating window remains active. Click in the Tools window. The computer beeps. Although you have clicked in another window, the regular window remains active. Play with the Tools window. Move it around. Click in the close box, and the window disappears. Show it again, then choose **Hide Tools** from the **Window** menu. The window hides.

Create several new regular windows. Look in the **Window** menu, and observe the various menu items. Close some windows, and observe what happens in the **Window** menu. Every time you open or close a regular window, the menu adjusts accordingly.

Activate various windows and observe the check mark in the **Window** menu. The mark should always denote the currently active regular window. Try the command keys for **Window** menu items. The corresponding window should become active.

If you would like to expand on this application, here are some suggestions.

- Add a `ClickSelf()` function to the regular window, and make it beep just like the floating window. Rebuild the app, make two windows, and click the inactive window. The window activates, but does not beep. Click again in the active window, and it beeps. Now, use Constructor to turn on the GetSelectClick feature of the regular window. Repeat the ex-

periment. What happens this time? Does the window beep when you activate it? It should, because the activating click now activates the window and is passed to the window for processing.

- Turn off the GetSelectClick feature for the floating window and see if it makes a difference. It shouldn't. Why not? Because a floating window is always active.
- Create a function that allows the user to change a window title. Update the **Window** menu along with the window title.
- Put some content in the windows!

Congratulations! You've passed another milestone. You have now worked extensively with menus, and created and used both regular and floating windows. In the next chapter we cover another kind of window in great detail—dialogs.

However, before you go running off to the next stage of the adventure, we want you to stop for a moment and think about the concept of adding a **Window** menu to an application.

As implemented in this chapter, the menu is an integral part of the application. There is nothing wrong with that. However, adding such a menu to existing code or removing it from a project would entail some substantial changes scattered here and there in the code. For example, you would have to modify window creation, window destruction, the application object's `FindCommandStatus()` and `ObeyCommand()` functions, as well as the application constructor.

You might wish it could be easier than this. Well, it is. We are going to revisit this topic in Chapter 15, "Periodicals and Attachments." Keep it in mind.

12

Dialogs

In this chapter we discuss how PowerPlant handles dialog boxes. Creating and using a dialog in a PowerPlant application is simple. You'll learn just how easy it is as we discuss:

- What Is a Dialog—comparing traditional and PowerPlant dialog handling.
- Dialog Characteristics—the special features of LDialogBox that distinguishes it from other windows.
- Working With Dialogs—how to create and use all kinds of dialogs in a PowerPlant application.

Along the way we will also encounter some PowerPlant utilities that manage simple dialogs almost automatically. We will use the terms "dialog" and "dialog box" as synonyms.

What Is a Dialog

In this brief section we look at dialogs from three perspectives:

- Traditional Dialogs—the standard Mac OS approach to dialog management
- PowerPlant Dialogs—the PowerPlant approach to dialog management
- LDialogBox Hierarchy—the LDialogBox ancestors, and what they give to LDialogBox

When you're through this section you should have a clear picture of the PowerPlant "philosophy" on dialogs.

Traditional Dialogs

In traditional Macintosh programming, a dialog box is a special kind of window. If you have written dialog-box code, you know

that you must write special code to ensure that a dialog box behaves properly.

To use a modal dialog, you call the Toolbox routine `ModalDialog()`. You must provide an event filter procedure in the `ModalDialog()` call. If you do not, the modal dialog seizes complete control of the computer. Nothing can work in the background. Windows won't update, and background processes won't receive time. This is not good.

To manage movable modal and modeless dialogs, you must modify the event dispatch mechanism of your application. You use `IsDialogEvent()` to identify dialog-related events. You use `DialogSelect()` to take care of dialog-related events. The process occurs as an adjunct to your regular event handling. As a result, much of dialog event-handling code duplicates code in your main event handling mechanism.

Bottom line, in traditional Mac OS programming, a dialog is a special kind of window that require special event handling and dispatch. This is not true in PowerPlant.

PowerPlant Dialogs

In terms of event handling and dispatch, there is no distinction in PowerPlant between a dialog and any other kind of window. In PowerPlant, a dialog is just another kind of window. Sure, it is a special kind of window, but the code to handle the dialog is built right into the PowerPlant event handling and dispatch mechanism.

In PowerPlant, a modal window occupies a special layer in the desktop display. As a result, other windows are deactivated properly and the modal window (or an object in the window) gets first crack at an event. There is no need for DLOG or DITL resources. Constructor makes it easy to arrange controls in any window.

The PowerPlant command and visual hierarchies work together to ensure that whatever event occurs, the appropriate pane, view, or control receives the event and has an opportunity to process it. The fact that the target object or pane is in a dialog window is irrelevant to this process.

PowerPlant does not use `ModalDialog()`, `IsDialogEvent()` or `DialogSelect()` to identify and dispatch dialog-related events. When a PowerPlant application receives a command or keystroke, the current target object gets the event, whatever kind of window contains the target object. When a PowerPlant application receives a click, the pane clicked gets the event. If the click is outside a modal dialog, the application beeps. Everything works just the way it should, with the same dispatch mechanism used throughout the application for all windows: movable modal, floating, or regular.

However, there is one limitation. If you want an old-fashioned, non-movable modal dialog, you must use the Mac OS Toolbox and `ModalDialog()` and an event filter. You can do this inside a PowerPlant application, as you'll see in this chapter. However, you cannot use a PPob resource and PowerPlant event dispatch to manage a non-movable modal dialog.

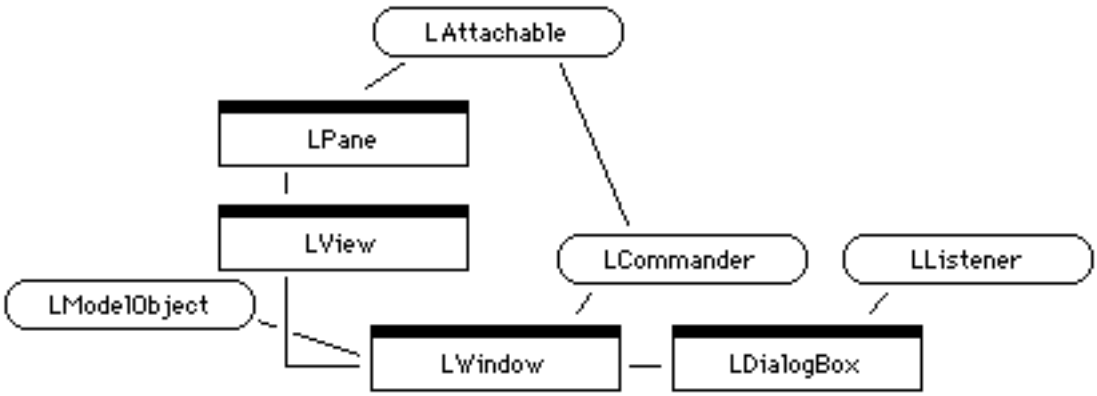
While it is important that you understand these design principles so you can use PowerPlant effectively, you also need to know the code-level implementation of this design. For that, we start with a quick look at the LDialogBox class hierarchy.

See also "Window layers" on page 354.

LDialogBox Hierarchy

Figure 12.1 illustrates the classes from which LDialogBox inherits.

Figure 12.1 LDialogBox hierarchy



You can see that an LDialogBox object is a window (hence a view and a pane), can have attachments, is an LModelObject so it is scriptable, is a commander, and is a listener. In a nutshell, an LDialogBox is a window that is also a listener.

Inheriting from LListener is the most significant distinction between an LDialogBox and other windows. Most dialog boxes contain control items. Because it is a listener, the dialog box object can listen to messages from its controls. It can manage the controls or perform other actions in response to the messages.

When you write dialog-related code, you can keep responsibility for managing the dialog right where it belongs—in the dialog itself, or in a supercommander.

Dialog Characteristics

LDialogBox is actually a very simple extension of LWindow. Because a dialog is a window, everything you learned about windows in the previous chapter applies to dialogs.

There is only one characteristic added to a regular window—button tracking for the default and cancel buttons.

LDialogBox has data members to identify the default and cancel buttons in a dialog. They are `mDefaultButtonID` and `mCancelButtonID`. You specify the button by its Pane ID number.

You would typically specify these values in Constructor when you build the visual hierarchy. When the LDialogBox stream constructor builds the dialog, it automatically puts the required outline (using LDefaultOutline) around the default button. LDialogBox also automatically supports the standard key-equivalents for both the default and cancel buttons—Enter or Return for the default button, command-Period or Cancel for the cancel button. You can see how in `LDialogBox::HandleKeyPress()`.

You can also set the buttons at runtime. The accessors for these data members are `SetDefaultButton()` and `SetCancelButton()`.

Examine `LDialogBox::FinishCreateSelf()` to see how PowerPlant sets up the buttons to behave correctly.

Working With Dialogs

For all its simplicity, PowerPlant gives you several different techniques you can use to create and manage a dialog. In this section—the real meat of this chapter—the discussion covers these topics:

- Creating a Dialog—using Constructor, creating a dialog on the fly, and deriving dialog classes.
- Messages in Dialogs—using messages generally, and using LDialogBox with negative message numbers.
- StDialogHandler—using this class to create and manage dialogs.
- Simple Movable Modal Dialogs—using UModalDialogs utility functions to run simple movable modal dialogs.
- Traditional Dialogs—how to integrate the Dialog Manager with PowerPlant to do things the old-fashioned way.

As we talk about dialog management, you will learn everything you need to know to implement a dialog by any of the five methods alluded to above:

- Use a class derived from LDialogBox.
- Use LDialogBox with negative message numbers.
- Use the StDialogHandler utility class.
- Use UModalDialog functions.
- Use the Mac OS directly.

You will find that one approach might be very useful in certain circumstances, and another useful at other times. But first, let's look at how to create the dialog object.

Creating a Dialog

You can create a dialog using Constructor, or on the fly in your code. We'll talk about each method. Then we discuss what you do when you derive your own class from LDialogBox.

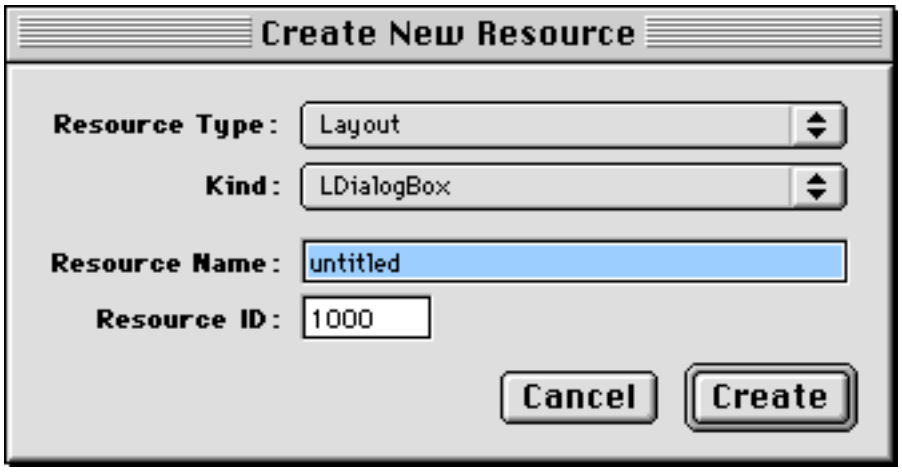
Using Constructor

Dialogs work almost exactly like windows.

If you have built a PPob resource for a dialog in Constructor, creating the dialog (and all of its contents) is simple. You call `LWindow::CreateWindow()`. You provide the resource ID number for the PPob resource, and a pointer to the dialog's supercommander. PowerPlant does the rest. There is no `LDialogBox::CreateDialog()`. The LWindow function serves just fine. Typically you typecast the returned LWindow pointer to be an LDialogBox pointer.

Creating a dialog in Constructor is simple. While in the Constructor project window, select the Windows and Views choose **New Resource** (command-K) from the **Edit** menu. When you do, the dialog in Figure 12.2 appears.

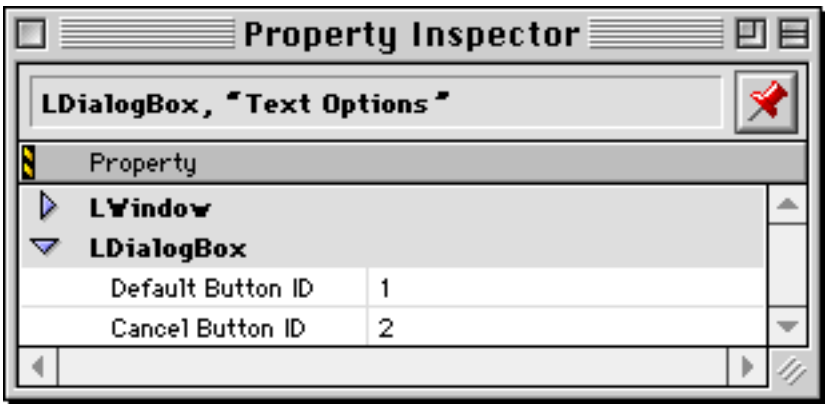
Figure 12.2 Creating a new dialog



Choose PPob as the resource type, and LDialogBox as your view type. You can set the name and ID right here. Click the create button to create the new PPob resource. Open the new PPob resource to see

the layout editor, and then the Property Inspector window for this particular dialog. It is identical to the window dialog as shown in Figure 11.4 on page 363, with one exception. You can specify the default and cancel buttons by Pane ID, as shown in Figure 12.3.

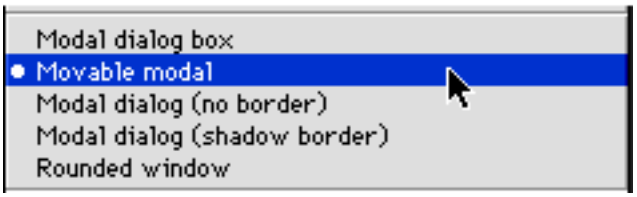
Figure 12.3 Tracking dialog buttons



Remember, when you derive your own classes you must change the class ID to your own unique value and register the class with PowerPlant before creating any objects of that class.

When setting the characteristics for a dialog window, set the Window Proc to one of the available dialog options, as shown in Figure 12.4.

Figure 12.4 Window Proc options



To create a movable modal dialog, choose the Movable modal item in the popup menu.

WARNING! Although you can choose a modal dialog box with various borders, the behavior of a pure modal dialog created this

way will not follow human interface guidelines. See "Traditional Dialogs" on page 410 for more on this topic

To create a modeless dialog, use the document window option. A modeless dialog should have a go away box, a title bar, no zoom box, no grow box, and should not be resizable.

Because of the elegant design of PowerPlant event handling, you don't need to do anything else to specify a movable modal or modeless dialog. These are just other kinds of windows. Of course, you must populate the dialog with the necessary panes, views, and controls, just like you would for any window.

See also "Creating a Window" on page 362 and "Register PowerPlant Classes" on page 288.

Creating a dialog on the fly

You build a dialog on the fly the same way you would a regular window. There is a dialog constructor that takes an `SWindowInfo` structure as its only parameter.

Typically, Constructor or the dialog utility functions available in PowerPlant will provide all the functionality you need. We discuss dialog utilities in "StDialogHandler" on page 408 and "Simple Movable Modal Dialogs" on page 410.

See also "Creating a window on the fly" on page 366.

Deriving your own dialogs

Deriving your own dialog class is one of the principal techniques used in PowerPlant for creating and managing custom dialogs. Each dialog is likely to have a different set of controls, and you may wish to respond to those controls in unique ways.

You can do so by deriving from LDialogBox and overriding the `ListenToMessage()` and `ObeyCommand()` functions. You are already familiar with these functions from our earlier discussions of listeners and commanders. LDialogBox is both a listener and a commander.

A typical approach to dialog management in a derived class is to perform the following steps.

1. Create the derived dialog object. Usually you do this with a PPob, after registering the new class with PowerPlant.
2. Link the dialog object to its controls. You call `LinkListenerToControls()` with the RidL resource ID for this dialog. Remember from the Controls chapter that Constructor automatically creates a RidL resource with the same resource ID as the window for all the controls in a window. Alternatively, you link a few individual controls to the dialog as necessary.
3. Override `ListenToMessage()`. The dialog object listens to messages from linked controls. `ListenToMessage()` can respond to the message directly. Or, `ListenToMessage()` can call the dialog's `ProcessCommand()` function. This function in turn calls `ObeyCommand()`.
4. Override `ObeyCommand()`. If `ListenToMessage()` calls `ProcessCommand()`, control passes to `ObeyCommand()`. Handle the message in `ObeyCommand()`.

Of course, you may override any other functions necessary to fully implement your dialog's behavior. You might override `FindCommandStatus()` to handle menu updating, for example.

You also add whatever new functions you need to process the information generated in the dialog. For example, you might define a function to manage enabling or disabling a set of controls that are dependent upon the state of a check box.

Taking this approach, responsibility for handling dialog-related events rests primarily with your dialog object. Your dialog's behavior is certainly message-based, and could be command-based as well if you transform messages into commands and call `ProcessCommand()`.

There is an alternative to putting all this responsibility down in the dialog object. For example, you might have a situation where changing a setting in a dialog might affect several windows. Responsibility for such a widespread change should probably reside with the application. It should almost certainly reside with some object

higher than the dialog. In such a case, the dialog should issue a command when it hears the message.

How do you send a command up through the chain of command?

Messages in Dialogs

Recall that a control can send any message by calling `BroadcastMessage()` at the appropriate moment. Some controls also use the `BroadcastValueMessage()` function. Either way, if the dialog is linked to the control, the dialog hears the message.

In response to the message, you want something to happen. As we discussed above, the dialog object can send a command to itself in response to the message by calling `ProcessCommand()`. The dialog can, alternatively, send a message/command directly to the dialog's supercommander. `LDialogBox::ListenToMessage()` has a mechanism for passing a command to a higher level. Listing 12.1 contains the code.

Listing 12.1 Excerpt from LDialogBox::ListenToMessage()

```
else if (inMessage < 0)
{
  // Relay message to supercommander
  if (GetSuperCommander() != nil) {
    SDialogResponse theResponse;
    theResponse.dialogBox = this;
    theResponse.messageParam = ioParam;
    ProcessCommand(inMessage, &theResponse);
  }
}
```

If the message is negative, LDialogBox creates an `SDialogResponse` structure. It then sends the message as a command to the supercommander's `ProcessCommand()` function, along with the `SDialogResponse` data. It is the supercommander's responsibility to handle the command or pass it on up the chain of command.

Keep in mind that there is a distinction between the *message* that a dialog receives, and a *command* that might be issued as a result of receiving the message. However, in `LDialogBox::ListenToMessage()` that distinction becomes blurred. The negative message number becomes a negative command number.

If you use this technique exclusively—that is, if all the messages (except the close message) from your dialog's controls are negative—then you don't have to derive a class from LDialogBox at all. You can give the dialog's supercommander the responsibility for responding to messages.

A third solution is a mix of the two approaches. You derive a dialog class from LDialogBox. However, if the dialog engenders a situation that requires high-level attention, you make sure that the control involved sends a negative message. You also ensure that somewhere in the chain of command there is a commander that can deal with that message appropriately.

Of course, if you override LDialogBox you can override `ListenToMessage()` and replace or ignore the negative message mechanism entirely.

Let's stop for a moment and consider all the possibilities. The situation is this: you have a dialog, it contains controls, the controls send messages. So far you have at least four possible ways of handling messages in a dialog.

- You create controls that are also listeners, and respond directly to each other. (We discussed this possibility in "Deriving your own controls" on page 224.)
- Your controls send positive message numbers, and the dialog object handles them in the `ListenToMessage()` or `ObeyCommand()` functions.
- Your controls send negative message numbers exclusively. You use LDialogBox directly, and require the supercommander to handle the resulting commands.
- Your controls send a mix of positive and negative messages. You distribute responsibility for responding to the messages between a dialog object and its supercommander.

The various PowerPlant mechanisms, while simple in their own rights, give you many possible paths to a desired outcome. As a PowerPlant programmer you analyze your needs and pick the path that best suits your situation.

TIP: When it comes time to close a dialog, you have choices again. The default or cancel buttons can send the dialog a `cmd_Close` message. Or you may call the dialog's `DoClose()` function directly at the appropriate moment. However, in some cases it is wise to simply hide the dialog and show it again when necessary. To implement this, you should override `AllowSubRemoval()` to hide the window rather than deleting it, and return false to not allow removal of the dialog. Overriding `AllowSubRemoval()` ensures that the correct behavior occurs whether `AttemptClose()` or `DoClose()` is called.

While everything up to now may seem complicated, it's about to get a lot simpler.

See also "Broadcasting" on page 227.

StDialogHandler

StDialogHandler is a stack-based utility class for managing movable modal dialogs.

Until now, the various strategies we've discussed for managing a dialog all rely on the PowerPlant event dispatch mechanism. StDialogHandler takes a different approach. It manages all events while a dialog box is active. This is closer to the traditional way in which the Mac OS handles dialogs.

Being a stack-based class, you create a local StDialogHandler object. The constructor requires that you pass in two parameters, the PPob resource ID for the dialog, and the supercommander for the dialog. The constructor requires that there be a RidL resource describing the controls in the dialog, and the RidL resource must have the same ID number as the PPob resource ID.

WARNING! Don't use StDialogHandler with a PPob for a non-movable modal window. The modal window will not behave according to human interface guidelines—the user will be able to switch applications. PowerPlant uses movable modal windows.

Before showing the dialog, you may need to set the values of any panes that could not be set in the PPob. After you display the dialog, you repeatedly call the StDialogHandler's `DoDialog()` function. This is analogous to the Toolbox `ModalDialog()` call. `DoDialog()` retrieves and processes events, and returns messages from the controls (all messages, positive or negative). In response to the message, your code acts accordingly. Listing 12.2 shows the code for a very simple loop that calls `DoDialog()` and responds to the messages received.

Listing 12.2 Sample DoDialog() loop

```
while (true) {
  MessageT hitMessage = theHandler.DoDialog();

  if (hitMessage == msg_Cancel)
  {
    break;
  }
  else if (hitMessage == msg_OK)
  {
    // process result of dialog
    break;
  }
}
```

When you are through with the dialog, you exit the `DoDialog()` loop, and ultimately exit the function in which the StDialogHandler object was created. The class destructor is called automatically and cleans up for you.

If that's not easy enough, it gets easier still.

Simple Movable Modal Dialogs

There are many cases where you want to display a movable modal dialog that gets a single number or string. For example, you may want the user to specify a font size, or enter a name.

The UModalDialogs class implements two static functions that use StDialogHandler. One of these functions returns a single number. The other returns a single string. Using them is trivial.

To display a dialog to get a single integer, call `UModalDialogs::AskForOneNumber()`. It handles all the work for you. You provide the supercommander, the PPob ID number, the pane ID number for the editable text field, and the default number that should appear in the dialog. Making the call displays a movable modal dialog for entering a single number. The call returns true if the user clicks OK. Here's an example of code that uses `UModalDialog::AskForOneNumber()`.

Listing 12.3 Getting a single number

```
Boolean result = UModalDialogs::AskForOneNumber (
          this, dialogID, editFieldID, number);
if (result)
{
  // do something with number
}
```

The `UModalDialogs::AskForOneString()` function works exactly the same, except you provide a string instead of a number.

TIP: Examine these two functions to learn how to use StDialogHandler effectively in more complex cases.

Traditional Dialogs

If you have a choice, do not use the Dialog Manager directly from PowerPlant. PowerPlant is much more powerful, and much more

elegant. The StDialogHandler and UModalDialogs classes really make dialog management a snap.

From time to time, however, you may find yourself forced to use the Dialog Manager directly or indirectly for various non-movable modal dialogs. You may wish to display a simple alert. You may need to support legacy code that uses `ModalDialog()`. Or you may want to do something as simple as display the Mac OS standard file dialog. You can do so, as long as you keep two things in mind.

First, remember that any time you display an alert or a modal dialog using the Dialog Manager, `ModalDialog()` seizes control of all events. PowerPlant event processing—such as giving time to periodicals—is interrupted. In addition, background processes get no time unless you provide an event filter.

Second, before doing anything that invokes the Dialog Manager from a PowerPlant application, you must call `UDesktop::Deactivate()`. This call ensures that all PowerPlant windows are properly deactivated before the modal dialog appears. After you dismiss the dialog, call `UDesktop::Activate()` to restore all PowerPlant windows to their correct state. You'll do this in the code exercise for this chapter.

TIP: The need to deactivate PowerPlant windows can sneak up on you. Remember to call `UDesktop::Deactivate()` before making any Mac OS Toolbox call that displays a dialog. Afterwards, call `UDesktop::Activate()`.

Summary

In this chapter you learned how PowerPlant handles dialog-related events. PowerPlant treats dialogs as just another kind of window and relies on the command and visual hierarchies to dispatch an event or command to the proper object, regardless of the nature of the window containing the object.

You also learned what makes a dialog object different from other windows—it can track the default and cancel buttons.

Finally, you discovered the flexibility PowerPlant gives you for dialog management. You can create movable modal, and modeless dialogs with ease. You can choose from a variety of methods for managing the dialog, including deriving your own dialog classes, using negative command numbers, using StDialogHandler, using UModalDialogs, and even the Mac OS Dialog Manager.

In the code exercise you can put this knowledge to practical use.

Code Exercise

In this code exercise you build an application named "Dialogs." This is a long code exercise. To make it a little more digestible, we're going to treat this as two separate code exercises: one for simple dialogs, and one for a more complicated dialog.

In the simple dialog exercise, you create and use four dialogs using three techniques: UModalDialogs, StDialogHandler, and the Mac OS Toolbox. In the complex dialog exercise, you create a dialog window that listens to its controls and handles the messages it receives in a variety of ways.

The application is based on the same dynamic caption object you used in the code exercise for Chapter 10, "Commanders and Menus." The four simple dialogs allow the user to change the text, font, font size, and style of the caption by modifying the caption's text traits resource or its descriptor. The complex dialog performs all these services in the same dialog. Changes in any dialog are applied to the caption text in the top regular window.

The Simple Dialog Interfaces

All the necessary resources have been provided for you in their entirety. There are four simple dialogs.

Table 12.1 Simple dialogs and their resources

Dialog	Type	Resource Type	Resource ID
Set Text	movable modal	PPob	1200
Set Font	movable modal	PPob	1300
Set Size	movable modal	PPob	1400
Set Style	modal	DLOG and DITL	1500

The PPob resources are in `Dialogs.ppob`. The DLOG and DITL resources are in `Dialogs.rsrc`. Constants for these resource IDs and the dialog contents are declared in `DialogsConstants.h`.

Explore these resources until you are comfortable with their elements. Each PPob contains a dialog window. When you examine the dialog characteristics, you'll see that each dialog is movable modal, and in the modal layer. Each dialog has an OK and a Cancel button, and one other item, either a popup menu or an editable text field.

Implementing Simple Dialogs

Figure 12.5 illustrates the application's **Dialog** menu. This part of the exercise concerns the last four items in the menu. These are the four simple dialogs listed in Table 12.1.

Figure 12.5 The Dialog menu

When the user chooses one of the last four items, the application object displays the appropriate dialog. When the user approves the di-

alog, the application object retrieves the necessary information and changes the topmost caption. If the user cancels the dialog, the application does nothing.

In this section you write the code to implement the text dialog using UModalDialogs. You implement the font dialog using StDialogHandler. The size dialog also uses StDialogHandler, so that code is provided for you. You implement the style dialog using the Mac OS Dialog Manager.

1. **Examine commands in the Dialog application.**

`ObeyCommand()` `CDialogsApp.cp`

In this step you explore `ObeyCommand()` so you have an idea of what's happening in support of the dialogs you are about to build.

This function has a series of `case` statements for the commands the application handles. Of particular interest to us here are these four commands:

- `cmd_SetTextDialog`
- `cmd_SetFontDialog`
- `cmd_SetSizeDialog`
- `cmd_SetStyleDialog`

The application receives these commands when the user chooses the corresponding item in the **Dialog** menu. Locate the `case` statements for these commands, and examine the code. You don't write any code in this step, just review the existing code.

Each starts identically. The application gets the top regular window. If there is a window and it is the right kind of window, the application then gets the dynamic caption object.

Listing 12.4 Setting up to create a dialog.

```
// Get the top regular window.
LWindow *theWindow = UDesktop::FetchTopRegular();

if ( theWindow != nil && theWindow->GetPaneID() ==
                rPPob_SampleTextWindow ) {
```

```
  // Get the dynamic caption.
  CDynamicCaption *theCaption;
  theCaption = dynamic_cast<CDynamicCaption *>
              (theWindow->FindPaneByID( kDynamicCaption ));
Assert_( theCaption != nil );
```

After it has the caption, the application gets either the caption descriptor (for the Set Text dialog) or the caption's text traits record (for the other dialogs).

Finally, in each `case` there is code that calls a function to manage the dialog. For example, to set text the code is:

```
if ( AskForText( theText ) ) {
  // Set the caption's text.
  theCaption->SetDescriptor( theText );
}
```

Each of the `AskFor...()` functions returns a boolean value indicating whether the user clicks OK or Cancel. If the user clicks OK, the application sets the appropriate information for the caption. In the example above, it sets the caption's descriptor.

This dispatch code is provided for you. You have already worked extensively with the `ObeyCommand()` function in previous exercises, so you know how it works. In the next steps you write three of the `AskFor...()` functions to display and manage dialogs. Each demonstrates a different technique for creating, displaying, and managing dialogs.

2. Use UModalDialogs to manage a dialog.

`AskForText()` `CDialogsApp.cp`

This movable modal dialog contains an LEditField pane to allow the user to enter text. Your task is to create, display, and manage the dialog. You get the text, dismiss the dialog, and return the correct value—true or false—to the caller. You can do all this with one line of code and the `UModalDialogs::AskForOneString()` function. The declared constant for the PPob resource is `rPPob_SetTextDialog`. The declared constant for the edit pane is `kSetTextEditField`.

```
  Boolean theResult = false;

  theResult = UModalDialogs::AskForOneString( this,
                rPPob_SetTextDialog, kSetTextEditField, ioText );

  return theResult;
```

3. Use StDialogHandler to manage a dialog.

`AskForFont()` `CDialogsApp.cp`

As you know, the StDialogHandler stack-based class takes over all event processing. Your tasks are similar to those in the previous step. You must create, display, manage, and dismiss the dialog. Along the way, you get the necessary data (in this case a font number) and return the correct value to the caller.

a. Create the dialog.

Declare a local StDialogHandler variable. You provide the PPob resource ID and the commander. The declared constant for the resource ID is `rPPob_SetFontDialog`. After you create the handler, get the pointer to the dialog window from the handler object.

```
Boolean theResult = false;

// Create the dialog handler.
StDialogHandler theHandler( rPPob_SetFontDialog, this );

// Get the dialog.
LWindow *theDialog;
theDialog = theHandler.GetDialog();
Assert_( theDialog != nil );
```

Existing code then gets the popup menu item and initializes it.

b. Show the dialog.

After the existing code initializes the popup menu, display the dialog. Solution code for this substep is in substep c.

c. Run the dialog.

After displaying the dialog, loop repeatedly and call the handler object's `DoDialog()` function. Respond to the messages. There are two possible messages, `msg_Cancel` and `msg_OK`.

If the user cancels the dialog, exit the loop and return a value of `false`.

If the user accepts the dialog, get the text for the current font in the popup menu. Use the Toolbox function `GetMenuItemText()`. Then get the font number. Use the Toolbox function `GetFNum()`. Set `theResult` to `true`, and return.

```
// Make the dialog visible.
theDialog->Show();

while ( true ) {

  // Handle dialog messages.
  MessageT theMessage = theHandler.DoDialog();

  if ( theMessage == msg_Cancel ) {

    // Just break out of the loop.
    break;

  } else if ( theMessage == msg_OK ) {

    // Get the font name chosen.
    ::GetMenuItemText( thePopup->GetMacMenuH(),
                       thePopup->GetValue(), ioFontName );

    // Get the font number.
    ::GetFNum( ioFontName, &outFontNumber );

    // Turn on the result flag and
    // break out of the loop.
    theResult = true;
    break;
  }
}
```

```
return theResult;
```

This approach gives you more flexibility than the UModalDialogs utilities. Although this is a simple example that receives two messages, you can use StDialogHandler to receive and handle an arbitrary set of messages from the dialog contents.

The code in `AskForFontSize()` uses the same approach as this step. That code is provided for you.

4. Use the Mac OS Dialog Manager to manage a dialog.

`AskForStyle()` `CDialogsApp.cp`

In this step you write code at the beginning and end of this function. The code to run the dialog is provided for you, because it is pure Toolbox and has nothing directly to do with PowerPlant.

However, if you wish to use the Dialog Manager to handle dialogs, there are some PowerPlant-related tasks you must perform before you run the dialog.

a. Deactivate PowerPlant windows.

Use a function in UDesktop.

b. Clear PowerPlant focus.

The code you are about to write sets the GrafPort directly, effectively changing the port behind PowerPlant's back. Call `LView::OutOfFocus()` so PowerPlant knows that the focus is no longer reliable.

```
DialogPtr theDialog;

// Deactivate desktop windows.
UDesktop::Deactivate();

// invalidate LView's focus cache.
LView::OutOfFocus( nil );
```

```
// Create the dialog.
```

Existing code creates and runs the dialog. It uses `ModalDialog()`. In this example there is no event filter proc provided. If you use the Dialog Manager and modal dialogs in a real application, you should provide an event filter.

After the dialog is complete, the existing code disposes of the dialog. You have one more PowerPlant task to perform.

c. Reactivate PowerPlant windows.

Once again, use a function in UDesktop. This code goes at the very end of the function.

```
::DisposeDialog( theDialog );

// Activate desktop windows.
UDesktop::Activate();
```

That takes care of the simple dialogs. Save your work. You can leave the file open if you wish, you'll be using it again a little later.

5. Build and run the application.

Make the application. You may get a couple of warnings about unused variables, but you can ignore them. You'll use them in the second part of this exercise.

All of the four simple dialogs should work correctly at this point. Display each dialog in turn.

Observe the state of the menu bar and menu items. When you display the Set Text dialog, the **Edit** menu is enabled. UModalDialogs takes care of that for you because you have an editable text field in the dialog. For the Set Font and Set Size dialogs, only the System menus—Apple, Help, Application—are available. For the Set Style dialog, the Apple and Application menus are also disabled. This is a modal dialog, and you cannot leave the application when a modal dialog is frontmost.

Play with the various dialogs, clicking the Cancel and OK buttons, and observe the effect on the sample text.

If the application does not behave as you expect, check your code against the solution code to make sure you performed each step correctly. If you'd like to study further, you can quit the application, enable the debugger, rebuild the application, and run it under the debugger. Set breakpoints at various judicious spots and watch what happens. You could also add a new dialog to modify the caption's color. Perhaps you can use the color control you built in the exercise on controls.

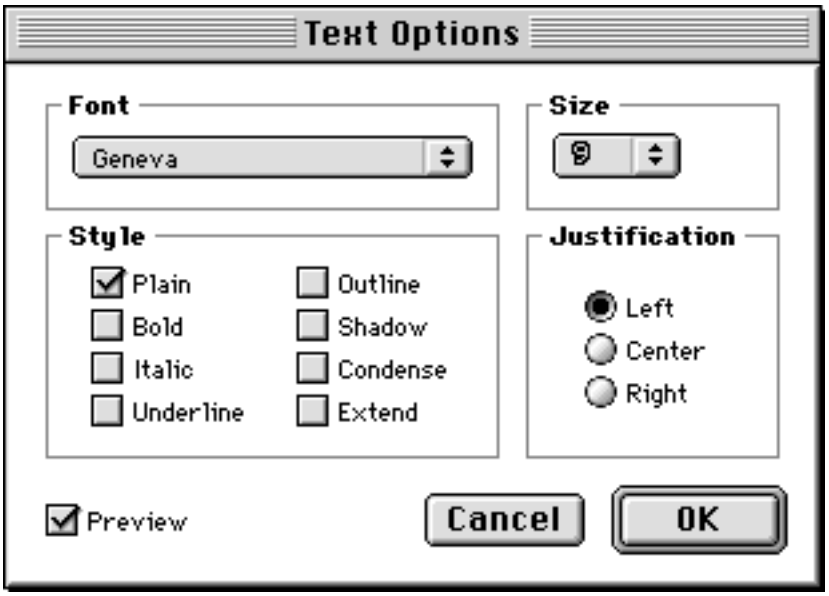
However, if you'd like to continue forward, there's more to be done! Let's put all four dialogs into one. This enables the user to modify any text trait in the same dialog, and to preview the effect of changes.

The Complex Dialog Interface

In essence, a dialog in PowerPlant is a window with controls. You use those controls just like you would in any other window. The controls send messages. Any appropriate object may listen to the controls. When it receives a message the listener acts. Typical listeners include custom controls in the dialog, a custom commander object, the dialog window itself, or the application object. In this application, the listener is the dialog window.

Figure 12.6 illustrates the Text Options dialog you work with in the remaining steps in this exercise. Use Constructor to open the `Dialogs.ppob` project file, and examine the Text Options Dialog PPob resource with ID number 1100. This PPob resource describes the contents of this dialog.

Figure 12.6 The Text Options dialog



If you examine the window characteristics you'll see it is a movable modal dialog in the modal layer. It is a custom object. The class ID is TxtD. The text options dialog class overrides a few of the LDialogBox functions. We'll examine the class in the next step.

After examining the window characteristics, look at the individual controls in the window. Each has a value message. The value message is the sum of the PPob resource ID and the pane ID. For example, the Preview check box is pane ID 3, and the value message is 1103. This kind of systematic numbering can make it easier to keep track of controls in a complex dialog.

The OK and Cancel buttons use negative message numbers, -1101 and -1102 respectively. We'll use this fact to demonstrate what happens to negative message numbers when working with a dialog.

Implementing a Complex Dialog

Like the simple dialogs, the user displays the Text Options dialog by choosing the correct item in the **Dialog** menu. The application's `ObeyCommand()` function receives a `cmd_TextOptionsDialog` message. That's where we'll begin writing code.

Before we do that, however, let's take a quick look at the CTextOptionsDialog class.

6. Examine the CTextOptionsDialog Class

`class declaration` `CTextOptionsDialog.h`

The class ID is TxtD, just like the value specified in the PPob resource.

This class has two new data members, `mTextTraits` and `mOriginalTextTraits`. The dialog modifies the text traits resource interactively, but must allow the user to cancel the operation and restore the caption to its original state. The dialog object stores a pristine copy of the original text traits in `mOriginalTextTraits`. The `mTextTraits` member is the "new" text traits. It reflects the changes going on because of choices in the dialog.

This class overrides:

- `FinishCreateSelf()`
- `ListenToMessage()`
- `FindCommandStatus()`

You'll work on each of these functions in subsequent steps.

This class also declares three new functions:

- `SetupDialog()`
- `GetTextTraits()`
- `AdjustSizeMenuForFont()`

Each of these functions is provided for you. `SetupDialog()` initializes the data members and sets the controls in the window to initial values. `GetTextTraits()` returns the modified or "new" text traits. `AdjustSizeMenuForFont()` sets the items in the Size popup menu to use outline style if there is a bitmap font available for that size.

In the remaining steps in this exercise we're going to take the following path. We start at the application and create the window. Then we write the code to implement the dialog. Finally, we return

to the application to handle messages not fully handled by the dialog. Let's get started.

7. **Instantiate and display the dialog window.**

`ObeyCommand()` `CDialogsApp.cp`

When the user chooses the **Text Options** item in the **Dialog** menu, the application's `ObeyCommand()` function receives a `cmd_TextOptionsDialog` message. There is a `case` statement to handle that message. The existing code gets the top regular window and the text traits for the caption object inside that window.

After that, you have three tasks to accomplish. You must:

a. Create the dialog.

Use the `LWindow::CreateWindow()` function. Typecast the return value as a CTextOptionsDialog pointer. The declared constant for the PPob resource ID is `rPPob_TextOptionsDialog`.

b. Initialize the dialog.

Use the dialog object's `SetupDialog()` function.

c. Display the dialog.

Use the dialog object's `Show()` function.

```
theCaption->GetTextTraits( theTextTraits );

// Create the text options dialog.
CTextOptionsDialog *theDialog;
theDialog = dynamic_cast<CTextOptionsDialog*>
         (LWindow::CreateWindow( rPPob_TextOptionsDialog, this ));
Assert_( theDialog != nil );

// Setup the dialog
theDialog->SetupDialog( theTextTraits );
```

```
// Show the dialog.
theDialog->Show();
```

Save your work. Notice that there is no event handling and no dialog loop. You are simply creating a window. The regular PowerPlant event handling mechanism takes care of all event handling.

8. **Finish building the dialog.**

   `FinishCreateSelf()` `CTextOptionsDialog.cp`

   After creating the dialog object, PowerPlant calls the object's `FinishCreateSelf()` function. You write the complete function in this step. This function should accomplish two tasks.

   a. Call the inherited FinishCreateSelf().

   CTextOptionsDialog inherits from LDialogBox. The inherited function sets up the OK and Cancel buttons.

   b. Link the dialog to its controls.

   Use `LinkListenerToControls()`. The declared constant for the RidL resource ID is `rRidL_TextOptionsDialog`.

```
// Call inherited. LDialogBox FinishCreateSelf
// sets up the default and cancel buttons.
LDialogBox::FinishCreateSelf();

// Link the dialog to the controls.
UReanimator::LinkListenerToControls( this, this,
                                     rRidL_TextOptionsDialog );
```

The dialog is now finished. In the normal course of events, the user clicks controls in the dialog window. The dialog's `ListenToMessage()` function handles each message. In the next few steps you handle some of those messages.

9. **Respond to the cancel message.**

   `ListenToMessage()` `CTextOptionsDialog.cp`

   When the user clicks the Cancel button, the dialog receives the `cmd_TxtD_CancelButton` message from the control. This message has a value of -1102. You must do two things.

a. Restore the original text traits.

Call the dialog's `ProcessCommand()` function. Send a `cmd_SetTextTraits` command along with the original text traits from the `mOriginalTextTraits` member. The application object ultimately receives and handles this command.

b. Dispose of the dialog.

This is an application-level responsibility. The application created the dialog, the application should dispose of it. Call the inherited `ListenToMessage()` function. The `LDialogBox::ListenToMessage()` function passes negative messages as commands to the supercommander, in this case the application.

```
case cmd_TxtD_CancelButton:
{
  // Restore the original text traits.
  ProcessCommand( cmd_SetTextTraits, &mOriginalTextTraits );

  // Pass message to inherited ListenToMessage.
  LDialogBox::ListenToMessage( inMessage, ioParam );
}
break;
```

10. Respond to the preview message.

`ListenToMessage()` `CTextOptionsDialog.cp`

When the user clicks the Preview check box, the dialog receives the `cmd_TxtD_PreviewCheckbox` message. In response you should:

a. Determine if the button is on or off.

Examine the contents of the `ioParam` parameter.

b. If the button is on, set the new text traits.

Call `ProcessCommand()`. Send a `cmd_SetTextTraits` command along with the new text traits from the `mTextTraits` member.

c. If the button is off, restore the original text traits.

Call `ProcessCommand()`. Send a `cmd_SetTextTraits` command along with the original text traits from the `mOriginal-TextTraits` member.

```
case cmd_TxtD_PreviewCheckbox:
{
  if ( *(static_cast<Int32 *>(ioParam)) == Button_On ) {

    // Use new text traits.
    ProcessCommand( cmd_SetTextTraits, &mTextTraits );

  } else { // Preview turning off

    // Restore original text traits.
    ProcessCommand( cmd_SetTextTraits, &mOriginalTextTraits );
  }
}
break;
```

11. Respond to the plain style message.

`ListenToMessage()` `CTextOptionsDialog.cp`

When the user clicks the plain style check box, you must not only set the text traits, but clear all the other style check boxes as well. However, every time you set the contents of one of the other check boxes, it sends a message! This could cause problems.

To complete this step you should:

a. Stop listening.

This prevents the dialog from hearing messages as it changes the values of certain controls.

b. Turn this check box on.

Use `SetValueForPaneID()`. The declared constant for this pane is `kTxtD_PlainCheckbox`.

c. Turn off all the other style check boxes.

Loop from `kTxtD_BoldCheckbox` to and including `kTxtD_ExtendCheckbox`. Turn each control off.

d. Start Listening.

Now that you are finished changing control values, you can listen again.

e. Set the new text traits style to normal.

Use `mTextTraits`. The Toolbox constant for this is `normal`.

f. If the preview feature is on, set the new traits.

Check the value of the `kTxtD_PreviewCheckbox` control. If it is on, send the `cmd_SetTextTraits` message.

```
case cmd_TxtD_PlainCheckbox:
{
  // Turn off listening temporarily.
  // Otherwise we get in a tug-of-war as
  // controls broadcast their changing values.
  StopListening();

  // Turn check box on always. You can't turn
  // this box off by clicking on it.
  SetValueForPaneID( kTxtD_PlainCheckbox, Button_On );

  for ( PaneIDT i = kTxtD_BoldCheckbox;
              i <= kTxtD_ExtendCheckbox; ++i ) {

    // Turn off the other style check boxes.
    SetValueForPaneID( i, Button_Off );
  }

  // Start listening again.
  StartListening();

  // Set the style to plain.
  mTextTraits.style = normal;

  if (GetValueForPaneID( kTxtD_PreviewCheckbox) == Button_On ) {
```

```
    // Send the set text traits command.
    ProcessCommand( cmd_SetTextTraits, &mTextTraits );
  }
}
break;
```

12. Pass unhandled messages up the chain.

`ListenToMessage()` `CTextOptionsDialog.cp`

The default `case` takes care of any messages not handled. They should go to the inherited `ListenToMessage()` function.

```
default:
{
  // Call inherited.
  LDialogBox::ListenToMessage( inMessage, ioParam );
  break;
}
```

The code to handle all the other messages is provided for you. Examine the code if you wish. The tasks performed are essentially the same as what you have already accomplished.

13. Update menus.

`FindCommandStatus()` `CTextOptionsDialog.cp`

The last thing the dialog object must take care of is menu updating. Here is an exception to the rule that you always pass on that which you don't handle to the inherited function. When a movable modal dialog is the frontmost window, the **Apple** menu, the **Help** menu, and the **Application** menu should be active. All others should be inactive. The System takes care of the **Help** and **Application** menus.

You should activate the About item in the **Apple** menu, and disable all other commands.

```
// Disable all commands.
outEnabled = false;

if ( inCommand == cmd_About ) {
```

```
  // Enable the about command.
  outEnabled = true;
}
```

Save your work and close this file. You have completely implemented the dialog window. The final task is for the application to handle those commands not taken care of by the dialog window.

14. Handle messages at the application.

`ObeyCommand()` `CDialogsApp.cp`

The application receives three commands from the Text Options dialog window:

- `cmd_TxtD_OKButton`—a negative message converted by `LDialogBox::ListenToMessage()` into a command and sent to the supercommander.
- `cmd_TxtD_CancelButton`—a negative message converted by `LDialogBox::ListenToMessage()` into a command and sent to the supercommander.
- `cmd_SetTextTraits`—a command issued by the dialog window's `ListenToMessage()` function to update the caption in the top regular window.

In this step you handle the cancel button. The remaining code is provided for you.

When the user cancels the Text Options dialog, the dialog sends a `cmd_SetTextTraits` command that restores the original text traits. The application's only duty is to dispose of the dialog window. To do that, you should:

a. Get a pointer to the dialog response record.

It's in the `ioParam` parameter.

b. Get a pointer to the dialog window.

This is stored in the `dialogBox` field of the dialog response record.

c. Delete the dialog.

Use the `delete` keyword.

```
case cmd_TxtD_CancelButton:
{
  // Get the dialog response.
  SDialogResponse *theResponse =
                        static_cast<SDialogResponse *> (ioParam);
  Assert_( theResponse != nil );

  // Get dialog box from the dialog response.
  CTextOptionsDialog *theDialog;
  theDialog = dyanmic_cast<CTextOptionsDialog *>
                       (theResponse->dialogBox);
  Assert_( theDialog != nil );

  // Delete the dialog.
  delete theDialog;
}
```

In response to a click on the OK button, the code provided for you uses the same technique to get the dialog window, and then get the new text traits from the window. The code then sets the traits and deletes the dialog.

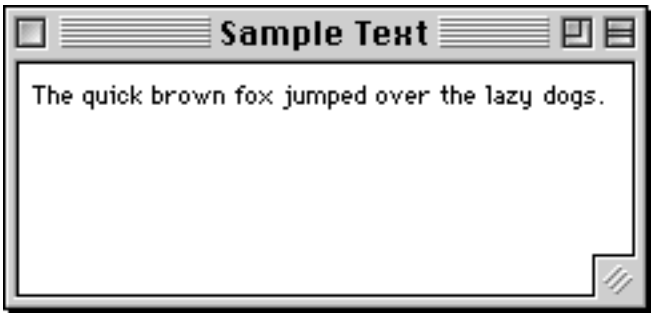
In response to the `cmd_SetTextTraits` command, the application gets the caption for the top regular window and sets its traits.

Save your work and close the file. You're all done.

15. Build and run the application.

At last! Make the project and run it. When you do, a window should appear with a caption object, as illustrated in Figure 12.7.

Figure 12.7 The dynamic caption window



Display the new Text Options dialog. Observe the state of the menu bar and menu items. Only the System menus—**Apple**, **Help**, **Application**—are available. This is appropriate for a movable modal dialog.

Change settings while the Preview check box is checked and unchecked. Observe the different behavior.

If the application does not behave as you expect, check your code against the solution code to make sure you performed each step correctly. If you'd like to study further, you can quit the application, enable the debugger, rebuild the application, and run it under the debugger. Set breakpoints at various judicious spots and watch what happens.

For example, set a breakpoint in the dialog's `ListenToMessage()` default case statement. Follow the flow of control when the OK button is clicked to see what happens to a negative message.

If you'd like to enhance the application, modify the dialog to include a color control, and change the text color.

Have a good time exploring!

Now it's time to leave the world of windows, dialogs, panes, views, and controls, and move on to a different part of PowerPlant: documents. In the next two chapters we explore the document pattern in PowerPlant. You'll learn how to open and close files on disk, write and read data in files, and print documents. The best part of the adventure lies ahead!

13

File I/O

You have a solid handle on how to manipulate the visual interface of a Macintosh application using PowerPlant. You have learned about all the visual elements: panes, views, and controls. You know how they communicate, and how to use them.

You have learned how to create a PowerPlant application, and how to use PowerPlant's debugging and memory management utilities. You know how to set up and manage the command structure in a PowerPlant application, and how PowerPlant handles menus. You have also learned how to manipulate the common interface views in a Mac application, windows and dialogs.

At this point, you can write a complete, self-contained PowerPlant application. However, there are two important features of an application that we have not discussed—file I/O and printing.

In this chapter we introduce you to PowerPlant's document-based strategy for managing persistent data—data that is saved to a file.

PowerPlant's approach to file I/O involves the collaboration of several different classes. As a result, it will help a great deal if, before we go into class-level details, we take a look at the big picture. With that in mind, the topics in this chapter are:

- The Document Strategy—how PowerPlant classes work together to save data.
- LDocApplication—the document services provided by LDocApplication.
- What Is a Document—a look at PowerPlant's document classes.
- What Is a File—the LFile class and how it serves as your bridge to the Mac File Manager.
- What Is a Stream—the various PowerPlant stream classes.

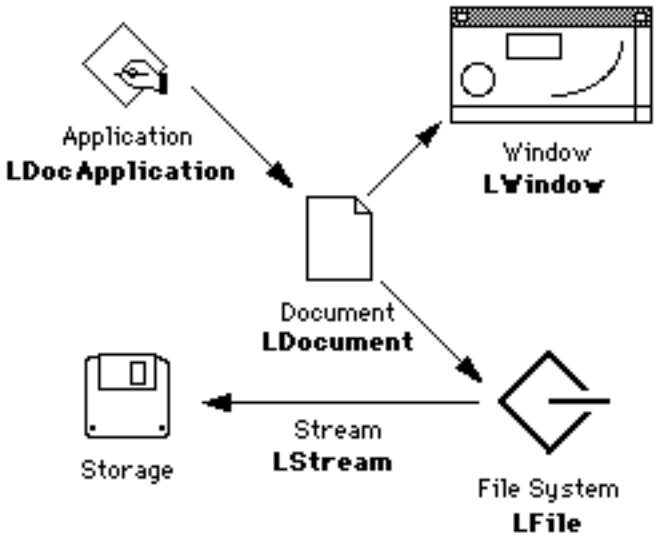
- Saving and Opening Files—how you can use all of these components together to read and write data.

The Document Strategy

In this section you get an aerial view of how PowerPlant manages documents. After you have the big picture, seeing how the pieces fit together will be a lot easier.

Figure 13.1 illustrates PowerPlant's approach to documents and file I/O. Each major component in the design—and its corresponding PowerPlant class—is shown. The discussion that follows explains all the details.

Figure 13.1 A document-centered design for file I/O



Until now we have ignored a simple fact about how computers work. A computer brings data into RAM for manipulation and display. It stores data "off site" on a hard disk or other storage medium. By "off site" we mean not inside the central processor or associated RAM. This design allows the data to survive while the computer is busy doing other things, or is turned off. In other words, the data becomes persistent.

We refer to an identifiable unit of stored data as a document. Essentially, this is what all of us call a "file" on our hard drives. We're going to use the term "document" in this discussion, to avoid confusion a little later on.

In a real sense, the application program itself is a document. The operating system loads it (or relevant parts of it) into RAM for manipulation and display.

However, we're going to take a somewhat more narrow view of a document. In this discussion, a document is an identifiable unit of stored data *used by an application*! The document contains whatever information is necessary for the application's purpose. It might be a single byte, or it might be gigabytes of information.

In this approach, the document is fundamental. It is *the* critical unit used by an application for data management.

Although most applications use documents, some do not. For the rest of this discussion we assume that the applications we speak of are all document-dependent.

An application that uses documents must have document management functions. The application must open a document, close the document, and create a new document. In PowerPlant, this behavior is encapsulated in the LDocApplication class.

In good object-oriented design, the document should be—as much as possible—responsible for itself. Then the application does not need to know the details of the document's internal workings.

Therefore, the document can be thought of as an independent object that collaborates with the application. In PowerPlant, this is the LDocument class, and its descendant, LSingleDoc.

The document is responsible for its own maintenance. Although the application issues commands to open and close a document, it is the document itself that is responsible for writing its contents to storage and reading its contents from storage. The application simply knows it has a document. The document knows what its contents

are. This frees the application from the responsibility of knowing anything about the contents of the document.

Because the document is responsible for reading and writing storage, the document must deal with the computer's file system. In PowerPlant, that interaction occurs through an LFile object. There is an important design subtlety here. The LFile object does not represent just the document on disk—what we commonly refer to as a file. The LFile object represents the access path to the document in storage. In a very real sense, the LFile object is the interface to *the file system itself*.

The document issues the commands, but the LFile object is responsible for the work when it comes time to actually read or write data in storage. There is more than one way to accomplish the task. You can read all the data in a single block. Or you can treat the stored information as a stream and read in the necessary bits and pieces. PowerPlant uses the LStream class to help you access data as a stream, should you choose to do so.

Of course, after you have moved the data out of storage and into RAM, it is quite likely that some person will want to view the data. That's where windows come into play. A window is merely the document's way of displaying the data on a monitor. In this strategy, a window is subservient to a document. The document "owns" the window that displays its data. The window has nothing whatsoever to do with saving contents to storage.

This is really the essence of the LDocument class. It associates files and windows.

Now that you know how the pieces fit together, you might want to take another look at Figure 13.1. Then we can talk about the individual elements in this design.

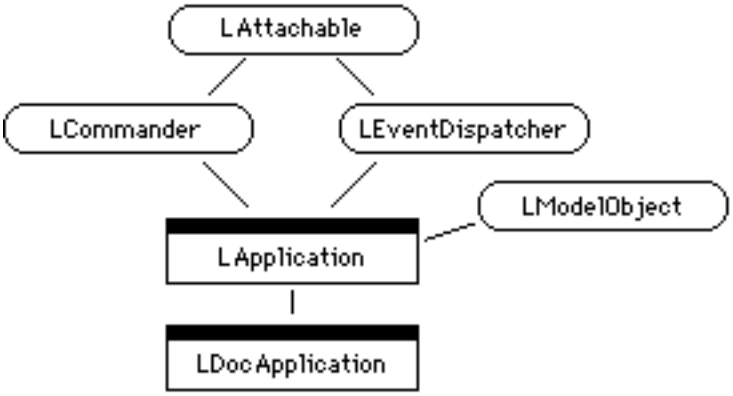
LDocApplication

The application forms the top of the document chain in terms of responsibility. It issues commands that open, close, and create new

documents. It is also responsible for issuing the command to print a document, but we talk about that in the next chapter.

LDocApplication is a simple extension of the PowerPlant LApplication class. Figure 13.2 illustrates its ancestry.

Figure 13.2 LDocApplication hierarchy



We discussed the principal features of an application in "The Application Object" on page 271. You can review that information if you need to brush up on the features of an application.

LDocApplication is simply an application with document support. It provides support on three levels:

- Command handling
- Apple events
- Function interface

When you create your own document-related PowerPlant application, you will subclass from LDocApplication. Although not an abstract class, several of the important member functions are empty.

Command handling

Because it is a commander, LDocApplication has the usual `ObeyCommand()` and `FindCommandStatus()` functions.

If you use standard PowerPlant command numbers for the **New**, **Open**, and **Page Setup** items, any class you derive from LDocApplication will handle these menu items and commands automatically.

In the default LDocApplication implementation, these items are always enabled. When the user chooses one of these commands from the **File** menu, `LDocApplication::ObeyCommand()` dispatches control to the appropriate function.

Apple events

LDocApplication has five functions that provide Apple event support for document-related commands. These functions are:

- `SendAEOpenDoc()`
- `SendAECreateDocument()`
- `DoAEOpenOrPrintDoc()`
- `HandleAppleEvent()`
- `HandleCreateElementEvent()`

These are handlers and dispatchers that manage the flow of control relating to documents. These are complete functions. It is unlikely that you will override them, with two exceptions. `HandleAppleEvent()` and `HandleCreateElementEvent()` are inherited from LModelObject. These two functions are commonly overridden when implementing scriptability.

Because PowerPlant document commands—even those from within the application itself—go by way of Apple events, these handlers suffice to dispatch all basic document-related actions.

However, these functions do not actually open, create, or print a document. They call implementation routines to do the actual work.

Function interface

LDocApplication provides these four functions for document management:

- `OpenDocument()`
- `PrintDocument()`

- `MakeNewDocument()`
- `ChooseDocument()`

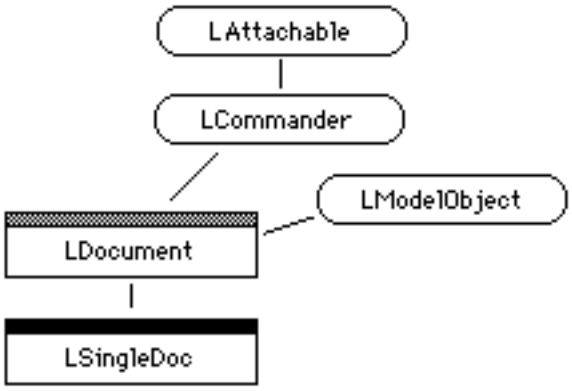
However, in LDocApplication, these functions do nothing. You must derive your own application class from LDocApplication, and write code to implement these four functions. We talk about how to do that in "Saving and Opening Files" on page 450.

Before that, let's look at the other components of the PowerPlant document design pattern.

What Is a Document

From the PowerPlant perspective, a document is an object that descends from LDocument. LDocument itself cannot be instantiated. Figure 13.3 illustrates the LDocument class hierarchy. LDocument appears with a grey bar because it is an abstract class.

Figure 13.3 LDocument class hierarchy



In this section we look at the characteristics and behaviors of both:

- LDocument
- LSingleDoc

LDocument

As an abstract class, LDocument provides a generic interface for all documents. LDocument has four data members of interest to us.

Table 13.1 LDocument data members

Data member	Stores
`mIsModified`	document has been changed
`mIsSpecified`	document is associated with a file on disk
`mPrintRecordH`	handle to Mac OS print record
`sDocumentList`	LArray object of all documents

The `sDocumentList` data member is static, hence a class variable. There is only one instance of this variable for all document objects.

The various LDocument member functions use these data members in their work. These functions provide support for:

- Command handling—how LDocument implements `ObeyCommand()` and `FindCommandStatus()`.
- Document management—LDocument's functions for saving and closing files.
- Other features—managing lists of documents and the document descriptor.

Command handling

Being a commander, LDocument has the usual `ObeyCommand()` and `FindCommandStatus()` functions.

If you use the standard PowerPlant command numbers for the **Close**, **Save**, **Save As**, **Revert**, **Print**, and **Print One** menu items, any class you derive from LDocument will handle these menu items and commands automatically.

In the default LDocument implementation, **Save As**, **Print**, and **Print One** are always enabled when there is a document active. The

Save menu item is enabled if there is a document active, and the document has been changed or doesn't yet exist on disk. The **Revert** item is enabled if the document has changed and already exists on disk.

When the user chooses one of these commands from the **File** menu, `LDocument::ObeyCommand()` dispatches control to the appropriate function in LDocument to implement the command. We talk about those functions in "Document management" below.

As a commander, LDocument has one more interesting behavior. It overrides the `AttemptQuitSelf()` function. This is where PowerPlant checks for any unsaved documents, and allows the user to save them before quitting. If you want to use a different alert than the PowerPlant default alert, or you want to manage this process in some other way, override this function in your own LDocument subclass.

Document management

LDocument is responsible for opening, saving, closing, reverting, and printing a document. Table 13.2 lists the member functions involved in this process.

Table 13.2 LDocument document management functions

Function	Purpose
`AskSaveAs()`	manage the standard file dialog
`DoAESave()`	do a Save As operation
`DoSave()`	do a Save operation
`DoAEClose()`	respond to a close Apple event
`AttemptClose()`	attempt to close the document
`DoRevert()`	revert to the last saved version
`DoPrint()`	print the document

In LDocument, the functions related to closing a file are complete. You probably won't need to override them. The `AskSaveAs()` function is also complete.

The other functions relating to saving, reverting, and printing a document are all empty. You must provide the functionality in your LDocument subclass. We talk about how to do that in "Saving and Opening Files" on page 450.

Remember, most of the dispatch and command handling functionality is provided by the application framework. All you have to do is provide the final function at the end of the chain that actually implements the required behavior. In the case of closing a document, the framework handles that for you too.

Other features

LDocument has two very useful utility functions for document management. Table 13.3 lists them.

Table 13.3 LDocument utility functions

Function	Purpose
`GetDocumentList()`	returns a reference to the LList object containing all open documents
`FindNamedDocument()`	returns an LDocument pointer to the specified document

These are both static functions, so you may call them at any time using the LDocument class specifier. These functions allow you global access to all open documents, or to a specific named document.

There is one other feature of LDocument. It has a descriptor characteristic. This is analogous to a pane's descriptor. However, the `GetDescriptor()` accessor is a pure virtual function. How you implement the descriptor feature in your own document class is up to you. Typically you would fill in the `outDescriptor` parameter with the name of the document.

LDocument is an abstract class, and as such cannot be instantiated. However, any class you derive from LDocument will have all of these features. PowerPlant derives one such class, LSingleDoc.

LSingleDoc

LSingleDoc is a very simple extension of LDocument. Although it is not an abstract class, the important functions for saving, reverting, and printing a document are still empty. You must derive your own class from LSingleDoc to implement real functionality.

The purpose of LSingleDoc is to provide the connection between an LDocument object, a single document on disk, and a single window on screen.

LSingleDoc has two new data members. They are `mWindow` and `mFile`. The `mWindow` data member is, predictably, a pointer to an LWindow object. The `mFile` data member is a pointer to an LFile object. You are already familiar with LWindow. We discuss LFile's characteristics and behaviors in "What Is a File" below.

There are no accessors for either data member. They are protected, so you can access them directly only from inside the class. You cannot access them from outside an LSingleDoc descendant.

In LSingleDoc, the `GetDescriptor()` function returns the name of the document. If the document is associated with a file on disk, it returns the name of the file. If there is no file, but there is a window (an unsaved new document for example), it returns the name of the window. If there is neither a file nor a window, it returns zero.

LSingleDoc provides one other useful feature for you. It overrides the LCommander `AllowSubRemoval()` method. An attempt to close a document-related window is really an attempt to close the document that owns the window. `LSingleDoc::AllowSubRemoval()` intercepts the command and attempts to close the document. If successful, the document will close the window as part of the process of closing itself.

What Is a File

The LFile object is a base class in PowerPlant. That is, it has no ancestor classes. In addition, it is an independent module in PowerPlant with no dependencies on other parts of the framework. You can use LFile independently, if you wish, as a wrapper for the Mac OS File Manager. See Figure 13.4 on page 447 for an illustration of the LFile class hierarchy.

Remember from our discussion of the PowerPlant file I/O design strategy that the LFile object represents both the document in storage on disk, and the file system itself. This duality is reflected in LFile attributes and LFile behaviors.

LFile attributes

LFile has three data members, with accessors for each. These data members store the Mac OS `FSSpec` record for the file on disk, the reference number (refNum) for the data fork of the file, and the refNum for the resource fork of the file. Table 13.4 lists the accessors.

Table 13.4 LFile data accessors

Accessor	Purpose
`GetSpecifier()`	return FSSpec for the file
`SetSpecifier()`	set the FSSpec for the file
`GetDataForkRefNum()`	return refNum for data fork
`GetResourceForkRefNum()`	return refNum for resource fork

In normal use, when you create an LDocument object for an existing file (when opening a file, for example), you create an LFile object and set the specifier to point to the correct file. The LFile object then becomes your path to disk. When you save a new document, you get a new specifier (typically obtained from the standard "put file" dialog). The LFile object takes care of setting the refNum values for the two file forks.

LFile behaviors

LFile also encapsulates everything you need for file creation, working with the data fork, and opening or closing the resource fork. This is the interface into the File Manager. Table 13.5 lists the functions in LFile.

Table 13.5 LFile functions

Function	Purpose
`MakeAlias()`	return an alias for specified file
`CreateNewFile()`	create a file with resource map and empty data fork
`CreateNewDataFile()`	create a file with data fork only
`OpenDataFork()`	open path to data fork
`CloseDataFork()`	close path to data fork
`ReadDataFork()`	read entire data fork
`WriteDataFork()`	write entire data fork
`OpenResourceFork()`	open path to resource fork
`CloseResourceFork()`	close path to resource fork

All of these functions are fully realized in LFile. These LFile wrapper functions greatly simplify the process of dealing with the Mac OS File Manager by hiding many of the details.

If you examine the source code for these functions, you'll see that they throw errors whenever necessary. You should catch these errors for any file operations you create, especially when reading and writing data.

There are no functions for reading or writing resources. You must write your own code to read or write data in the resource fork. You can use LFile to open and close the fork as necessary.

Finally, you may have also noticed that LFile reads and writes the data fork in one large chunk. If that data represents information for a variety of structures or objects, you must unflatten the data yourself.

While the one-gulp approach to the data fork works just fine in many cases, sometimes reading the entire data fork at once is neither wise nor possible. For example, large text documents, spreadsheets, databases, or images may overload your application's available memory. You may want to read just part of the data. That's where streams come into play.

What Is a Stream

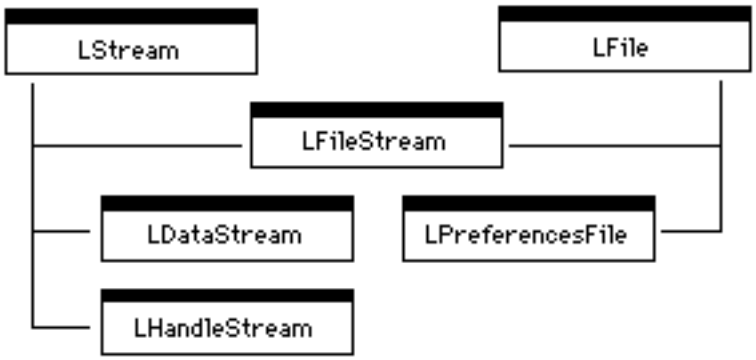
A stream is an ordered series of bytes. The stream concept is a very powerful one. When accessing data in a stream, it really doesn't matter where the data is coming from or going to. You have access to the stream, and you either read bytes from the stream or put bytes into the stream. While the ultimate source or destination of the stream is certainly of concern at one level, your data accessors don't need to know the source or destination. This keeps data I/O separate from reading and writing data.

PowerPlant provides general support for streams that you can use in any circumstance, and specific support for streams when accessing data in a file on disk. In this section we are going to concentrate on two classes:

- LStream
- LFileStream

Figure 13.4 illustrates how the PowerPlant stream and file classes are related.

Figure 13.4 LStream and LFile class hierarchy



Notice that LStream and its subclasses, like LFile and so many other segments of PowerPlant, form an independent module.

LStream

In PowerPlant, LStream is a generic implementation of streams for the Macintosh environment. It is not an abstract class, but it must be subclassed to be useful, as you'll see in a bit as we discuss:

- Stream attributes
- Stream behaviors

Stream attributes

A stream has a length and a marker. The length is the number of bytes in the stream. The marker is your current position in the stream—the point where the next byte is read from or written to. LStream defines accessors to these data members: `GetLength()`, `SetLength()`, `GetMarker()`, and `SetMarker()`.

You can measure from the beginning of the stream, the end of the stream, or from the position of the marker. PowerPlant defines an enumerated type, `EStreamFrom`, with these possible values:

- `streamFrom_Start`
- `streamFrom_End`
- `streamFrom_Marker`

Stream behaviors

The whole purpose of a stream is to read and write data, so all the functions are centered around that task. Table 13.6 lists the relevant functions.

Table 13.6 LStream data accessing functions

Function	Purpose
`WritePtr()`	write data referred to by a pointer
`ReadPtr()`	allocate a non-relocatable block and read data into it
`WriteHandle()`	write data referred to by a handle
`ReadHandle()`	allocate a relocatable block and read data into it
`WritePString()`	write a Pascal string
`ReadPString()`	read a Pascal string
WriteCString()	write a C string
ReadCString()	read a C string
`WriteData()`	write an arbitrary amount of data to the stream
`ReadData()`	read an arbitrary amount of data from the stream
`PeekData()`	read data without changing marker
ReadAll()	read remaining contents of stream into a handle

There are functions to read or write pointer data, handle data, Pascal style strings, and C strings. Each of these functions is fully realized in LStream, and you aren't likely to override them. Each reads or writes the length of the data first, and then reads or writes the correct amount of data in the stream.

In addition to these typical functions, LStream overloads the << and >> operators to read and write data (as in iostreams). There are overloaded versions of each operator to read or write the following data types:

- Pascal and C strings
- Rect
- Point
- the contents of a Handle
- signed and unsigned 8-, 16-, and 32-bit integers
- float and double numbers

Most of these functions—including the overloaded operators—rely on either the `ReadData()` or `WriteData()` routines. In LStream, these are empty functions. They read and write nothing. That's why you *must* subclass LStream for it to be useful. You can design the subclass to read or write data to the correct destination, be it a file system, a serial port, a network connection, and so on.

PowerPlant has three LStream subclasses.

- LDataStream is designed for a stream where the data buffer is a non-relocatable block—a pointer block.
- LHandleStream is designed for a stream where the data buffer is a relocatable block—a handle block. Consult *PowerPlant Revealed* and the source code for details on these classes.
- LFileStream is of special interest in a discussion of File I/O.

LFileStream

LFileStream inherits from both LFile and LStream. See Figure 13.4 on page 447. It has all the LFile behaviors we discussed earlier in this chapter, and overrides none of them.

It overrides and implements the `GetBytes()` and `PutBytes()` functions from LStream. LFileStream uses the File Manager calls `FSRead()` and `FSWrite()` to read or write the data. LFileStream also overrides the length and marker accessors to use the File Manager calls `GetEOF()`, `SetEOF()`, `GetFPos()`, and `SetFPos()`.

This is a perfect example of how to override LStream to customize it for a particular environment—in this case the File Manager.

If you use LFileStream as your document's file object, you can read or write data of arbitrary length in the data fork of the file. You can also get it all in one big block if you wish if you use the `ReadDataFork()` or `WriteDataFork()` functions inherited from LFile.

Saving and Opening Files

Now that we have had a good look at the pieces, let's see how they all fit together. In this section we discuss typical tasks you must perform as you:

- Implement an Application
- Implement a Document
- Implement a Preferences File

Implement an Application

Your first step in creating an application that can save and open files is to subclass LDocApplication. You override three functions: `MakeNewDocument()`, `OpenDocument()`, and `ChooseDocument()`.

These functions can be very simple. Listing 13.1 contains sample code for all three. The open and new routines rely on the document constructor to do most of the work. The routine to choose a document manages the `StandardGetFile()` dialog.

Listing 13.1 Sample document functions

```
void CMyApp::OpenDocument(FSSpec *inMacFSSpec)
{
  CMyDoc *theDoc = new CMyDoc(this, inMacFSSpec);
}

LModelObject* CMyApp::MakeNewDocument()
{
  CMyDoc *theDoc = new CMyDoc(this, nil);
```

```
  return theDoc;
}
void CMyApp::ChooseDocument()
{
  StandardFileReply macFileReply;
  SFTypeList        typeList;

  UDesktop::Deactivate();
  typeList[0] = 'TEXT';
  ::StandardGetFile(nil, 1, typeList, &macFileReply);
  UDesktop::Activate();
  if (macFileReply.sfGood) {
    SendAEOpenDoc(macFileReply.sfFile);
  }
}
```

Note the use of `UDesktop::Deactivate()` and `UDEsktop::Activate()` around the call to `StandardGetFile()`. Also, note that the code sends an Apple event to open the document, to support recordability in a scripting environment.

See also "Traditional Dialogs" on page 410.

Implement a Document

In your document class, you have a bit more work to do. Precisely what you do and how you do it will depend upon your application. However, the typical tasks are well defined. In this section we examine the tasks you perform when you:

- Create and open a document
- Save a document
- Revert a document

Create and open a document

In your document constructor—or in an initializing function called immediately after creating a document—you must set up all the information necessary for the document. Your tasks are:

1. Create a window for the document.

2. Set the window's name.
3. If you are opening a document, open the associated file.

You do not create a file object if you are creating a new document. The new document is not yet associated with a file on disk.

Opening a file also involves well-defined tasks. To open a file you:

1. Create and initialize a file object—either LFile, LFileStream, or a file object of your own design.
2. Open the data fork, read the data, and close the data fork. This assumes you use the data fork.
3. Open the resource fork, read resources, and close the resource fork. This assumes you use the resource fork.
4. Install the data content of your window.
5. Set the name of the window to match the file on disk.

The precise implementation of these steps will, of course, depend upon your application. For example, you may need to read resources first and data second. These steps are just a guide to the typical tasks you should keep in mind as you open a file.

Installing your data in the document may be an involved process. You decide your own data storage format. You may want to implement a streaming operation and rebuild objects from the stream. Or you may read the entire data fork at once, and then rebuild your document from the single block of data. Either way, remember that your reading and writing operations should be the exact converse of each other.

Having accomplished these tasks, you're still not quite through. You must save a file as well.

Save a document

To save a document, you implement the document class's `DoAE-Save()` and `DoSave()` functions. In the PowerPlant architecture, `DoAESave()` is called when the user is performing a Save As operation. This occurs when the user is saving a file for the first time, or saving the file under a new name.

The function receives a file specification and a desired file type. The file type is usually a constant, `fileType_Default`. This tells you to save the file in its "natural" file type for your application. However, the `inFileType` parameter may have a user-specified file type (to allow exporting a file in a different format, for example). A user-specified type might come from a custom save dialog, or from a script driving the application. If the file type is a non-default file type, specify the correct type and creator when saving the file.

To implement `DoAESave()` your function should:

1. Delete the document's existing file object. This does not delete the original file on disk, just the file object that represents the old file.
2. Create and initialize a new file object for the new file.
3. Create a new file on disk of the correct file type and creator.
4. Open the data and/or resource fork.
5. Save your document's data. In a well factored design, this usually means calling `DoSave()`.
6. Close the data and/or resource forks.
7. Set the window's name to the name of the new file.

To implement the `DoSave()` function, you perform these steps.

1. Gather your document's data.
2. Write it to disk.

Just what is involved in gathering your data depends upon your application. You may want to use a stream and write data for each object in the document. You may accumulate your data in a single block and write it all at once.

TIP: Recall that `LDocument::AskSaveAs()` manages the `StandardPutFile()` dialog when the user performs a Save As operation. This is the dialog the user sees when asked to provide a name for the file. If the user replaces the file, PowerPlant deletes the file on disk before beginning a new save operation. You may wish to override this mechanism in your own document class. A

more robust mechanism would be to create a temporary file and save the data to a temporary file. Only when the save operation is successful should you replace the existing file with the Toolbox call `FSpExchangeFiles()`.

Revert a document

The final I/O task your document must perform is reverting a file to its most recently saved state. Once again, the precise form this takes depends upon your application, but the steps are clear and simple. The document already has an associated file on disk. To revert the document you should:

1. Open the data fork, read the data, and close the data fork. This assumes you use the data fork.
2. Open the resource fork, read the resources, and close the resource fork. This assumes you use the resource fork.
3. Replace the existing contents of the document with the data.

The order of these tasks is significant. Don't delete the existing contents of your document until you've got the new contents from file. Then, if the operation fails you haven't destroyed the document.

WARNING! When creating a document-based application, don't forget to include the file `PP Document Alerts.rsrc` in your project. It contains PowerPlant's default document-related alerts. See Appendix B, "Resource Notes" for more information on the contents of this file.

Implement a Preferences File

Before we conclude this chapter, there is one more common, file-related task to discuss—maintaining a preferences file. Many applications have preference files.

PowerPlant makes creating and maintaining a preferences file as simple as possible with LPreferencesFile.

The constructor builds an FSSpec for a file in the Preferences folder in the System folder. That file has the name you specify. The constructor does not actually create or open the file on disk.

You use the member functions inherited from LFile to manage the file—create, open, read, write, and so forth. There is, however, one more nice feature to LPreferencesFile.

Preferences are often stored as a resource. The only new function in LPreferencesFile is `OpenOrCreateResourceFork()`. This function opens or creates the resource fork for the file, whichever is appropriate. After you have the fork open, you can use the Resource Manager to write your preferences resource to the file.

Summary

In this chapter you learned how several PowerPlant classes work together to implement a well-designed file I/O system.

LDocApplication is responsible for opening and creating documents—establishing documents in the application. In your application you implement simple functions to create and open a document.

LDocument is responsible for saving and closing documents, as well as reverting and printing—file management with documents. LSingleDoc implements the one document/one file/one window relationship typical of most Macintosh applications.

In your document class you write the routines to read the data from a file on disk into a document, and write data back out to disk. In the process you take advantage of LFile and LFileStream. These classes give you simple alternatives for reading and writing data to a Macintosh file.

Code Exercise

In this exercise you write an application named “Documents.” The Documents window contains an editable text pane, so you can type

text into the window. You did as much in the very first chapter in this book when you wrote PPEdit.

What's new here is that you can also open any text file (smaller than the 32K TextEdit limit), and save your work to a text file.

The Interface

The interface is not the center of attention in this exercise, but it is important. The PPob resource is provided for you. Take a moment to open the `Documents.ppob` project file with Constructor. Examine the PPob resource for the Documents window.

The window contains a scrolling view that contains a TxtV object. This is a custom object derived from LTextEditView.

The code for the CTextView class (class ID TxtV) is provided for you. Its added features include a "dirty" flag and an override of `UserChangedText()`. `CTextView::UserChangedText()` sets the dirty flag and the menu update flag. As a result, the application responds appropriately when the user changes the contents of the window.

For example, the Save item is enabled only if the document has been modified. The Revert item is enabled only if the document has a specified file and has been modified. This functionality is part of `LDocument::FindCommandStatus()`.

Other than that, there is nothing unusual in this window. You can close the PPob and quit Constructor.

Implementing Documents

Now it's time to write some code.

Not surprisingly, in the steps in this exercise you are going to accomplish the tasks outlined in this chapter for associating a window with a file. You create a document that keeps track of a window and a file, and move the data between them as appropriate.

Because this is a document-related application, notice that the project file includes the `PP Document Alerts.rsrc` file. If this file was not present, things would not go exactly as we planned.

You're going to work at this from the top down. That is, you start at the application level and implement the application functions necessary to support documents. Then you implement a document class derived from LSingleDoc. Let's get to it.

1. Examine the application class.

`class declaration` `CDocumentsApp.h`

Notice that this class inherits from LDocApplication, giving it all the default features of that class. It overrides five functions:

- `FindCommandStatus()`
- `StartUp()`
- `OpenDocument()`
- `MakeNewDocument()`
- `ChooseDocument()`

The `FindCommandStatus()` and `StartUp()` functions are provided for you. The `FindCommandStatus()` function disables the **Page Setup** item in the **File** menu, because this application doesn't support printing. We do printing in the next chapter. The `Startup()` function calls `ObeyCommand()` to create a new document.

You write the remaining functions in the next three steps. These are empty functions in LDocApplication.

2. Open a document.

`OpenDocument()` `CDocumentsApp.cp`

PowerPlant calls this function to open a new document. This may be in response to an open-document Apple event, or after the user chooses a document using the `StandardGetFile` dialog. The function receives a pointer to a valid `FSSpec` record containing the file specification.

In response, create a new document object. You must provide the document's supercommander and the file specification. You can do this with one line of code.

```
// Create a new document using the file spec.
new CTextDocument( this, inMacFSSpec );
```

3. Make a new document.

`MakeNewDocument()` `CDocumentsApp.cp`

PowerPlant calls this function when the user attempts to create a new document not connected to a specific file. The function receives no parameters, and returns a pointer to an LModelObject. The PowerPlant document classes inherit from LModelObject.

In response, create a new document object, and return a pointer to the object. You can use the same document constructor and pass nil for the file specification, if you design the constructor to handle both conditions (either a valid `FSSpec` pointer or nil). You'll do that in a subsequent step.

```
// Make a new empty document.
return new CTextDocument( this, nil );
```

4. Choose a document.

`ChooseDocument()` `CDocumentsApp.cp`

PowerPlant calls this function when the user chooses the **Open** item in the **File** menu. In response, you should display and manage the `StandardGetFile` dialog. To accomplish this task you should:

a. Deactivate the desktop.

Use the UDesktop utilities.

b. Allow the user to choose a document.

Declare an `SFTypeList` variable, and set its contents to look for files of type TEXT. Also declare a `StandardFileReply` variable. Then call `StandardGetFile()`.

c. Activate the desktop.

Use the UDesktop utilities.

d. Tell the application to open the file.

If the reply from `StandardGetFile()` is good, send the application an Apple event to open the document. The application object has a `SendAEOpenDoc()` function for just this purpose.

```
// Deactivate the desktop.
::UDesktop::Deactivate();

// Browse for a document.
SFTypeList theTypeList = {'TEXT'};
StandardFileReply theReply;
::StandardGetFile( nil, 1, theTypeList, &theReply);

// Activate the desktop.
::UDesktop::Activate();

// Send an apple event to open the file.
if ( theReply.sfGood )
  SendAEOpenDoc( theReply.sfFile );
```

You can save your work and close the file. That's all that's necessary at the application level to implement support for documents in PowerPlant.

However, the code you just wrote—as well as the rest of PowerPlant—relies on the existence of a fully-realized document class. You're going to implement a document class in the remaining steps in this exercise.

5. Examine the document class.

`class declaration` `CTextDocument.h`

When you look at the class declaration for this class, you see that CTextDocument inherits from LSingleDoc. This class implements the typical one-document-one-window model.

As a descendant of LSingleDoc, CTextDocument has data members `mWindow` for the window object, and `mFile` for the file object. It also

has a new data member, `mView`. This stores a pointer to the text view in the window so you can access the view directly.

The code for `FindCommandStatus()` is provided for you. It simply disables printing commands because this application does not support printing. The new function `NameNewDoc()` provides a properly-numbered "Untitled" title for new documents. The code is provided for you.

You write every other function in this class in the following steps. They are:

- `CTextDocument()`
- `OpenFile()`
- `DoAESave()`
- `DoSave()`
- `DoRevert()`
- `IsModified()`

6. Create a document object.

`CTextDocument()` `CTextDocument.cp`

This is the document constructor. This function receives a pointer to the supercommander, and a pointer to an `FSSpec` record. The second parameter may be nil.

This function has several tasks to accomplish. The existing code calls the LSingleDoc constructor with the supercommander parameter. In the body of this function, you should:

a. Create a window.

The constant for the PPob resource ID is `rPPob_TextWindow`. Store the resulting pointer in the `mWindow` data member.

b. Make the text view the latent subcommander.

The constant for the text view ID is `kTextView`. Get the pane, and store the pointer to the object in `mTextView`. Then make the text view the latent subcommander of the window object.

c. Name the window or open the file.

If `inFileSpec` is nil, call `NameNewDoc()`. If it is not nil, call `OpenFile()`. You write `OpenFile()` in the next step.

d. Show the window.

Use the window's `Show()` function.

```
// Create window for our document.
mWindow = LWindow::CreateWindow( rPPob_TextWindow, this );
Assert_( mWindow != nil );

// Make text view target when window activated.
mTextView = dynamic_cast<CTextView *>
              (mWindow->FindPaneByID( kTextView ));
Assert_( mTextView != nil );
mWindow->SetLatentSub( mTextView );

// Set name of window or open file.
if ( inFileSpec == nil ) {

  NameNewDoc();

} else {

  OpenFile( *inFileSpec );
}

// Make the window visible.
mWindow->Show();
```

In this function you have created a window and set the file (if necessary) for the document. These are the two primary features of a document in PowerPlant. However, the file-related tasks have been delegated to the `OpenFile()` function. That's next.

7. Open a file.

`OpenFile()` `CTextDocument.cp`

Opening a file is a process fraught with the possibility of error, so the code that attempts to open a file is written inside a `Try_` block.

The `Try_` and `Catch_` statements are provided for you. The code you write goes inside the `Try_` block.

This is a fairly complex step, but PowerPlant provides most of the functionality for you as part of the LFile and LTextEditView classes. The tasks you must accomplish are:

a. Create a new file object.

This doesn't create a file on disk, just the LFile object. Store the pointer to the object in the document's `mFile` data member.

b. Open the data fork of the file.

Use an LFile member function.

c. Read the entire contents of the file and close it.

Again, use LFile member functions. Get the data in a `Handle`. Declare a local `Handle` variable to store the handle. Don't forget to close the fork when you're done.

TIP: Actually, you might want to leave the file open to prevent the user from deleting the file in the Finder. If you do, make sure you close the file when the user closes the window.

---

d. Put the data in the text view and mark it as clean.

You have a pointer to the text view in the `mTextView` data member. Use an LTextEditView member function to set the contents of the CTextView object.

Also, because this is now a pristine view unchanged from the state of the file, set the text view's dirty flag to false. Use the text view's `SetDirty()` member function.

e. Dispose of the data.

Use the Toolbox call `DisposeHandle()` to dispose of the local `Handle`. LTextEditView makes its own copy of the data.

f. Set the window title to match the file name.

You have a pointer to the window object in `mWindow`. The name of the file is in the `inFileSpec.name` field.

g. Set the flag that says this document has a file.

Set the document's `mIsSpecified` data member.

```
Try_ {
// Create a new file object.
mFile = new LFile( inFileSpec );

// Open the data fork.
mFile->OpenDataFork( fsRdWrPerm );

// Read the entire file and close the file.
Handle theTextH = mFile->ReadDataFork();
mFile->CloseDataFork();

// Put the contents in the text view
// and clear the dirty flag.
mTextView->SetTextHandle( theTextH );
mTextView->SetDirty( false );

// Dispose of the text.
::DisposeHandle( theTextH );

// Set window title to the name of the file
mWindow->SetDescriptor( inFileSpec.name );

// Flag that document has an associated file.
mIsSpecified = true;

} Catch_( inErr ) {
```

Most of the details of the Mac OS File Manager have been hidden from you by PowerPlant. This is one of the advantages of an application framework.

You can now create and open a document fully and completely. You're making great progress. The next steps implement the ability to save a document.

8. **Implement "Save As" functionality.**

`DoAESave()` `CTextDocument.cp`

When the user chooses the **Save As** item in the **File** menu, or attempts to save a document that has no associated file on disk, control passes to this function. The function receives a file specification and a desired file type.

Again, there are several tasks you must accomplish.

a. Delete the existing file object.

This is the object pointed to by `mFile`. This does not delete any file on disk. You're just setting up a new file.

b. Make a new file object.

This doesn't create a file on disk, just the LFile object. Store the pointer to the object in the document's `mFile` data member.

c. Set the correct file type.

In this case, the default type is 'TEXT'. The `inFileType` parameter might be a user-specified type from a custom save dialog, if you support saving files in various formats.

Set a local `OSType` variable to 'TEXT'. If `inFileType` is not `fileType_Default`, set the `OSType` variable to `inFileType`.

d. Make a new file on disk.

Use an LFile member function. You must specify a file creator. You can use 'ttxt' if you wish, the creator for TeachText.

e. Write the data to the file.

Call `DoSave()`. You write this function in the next step.

f. Set the window title to match the file name.

You have a pointer to the window object in `mWindow`. The name of the file is in the `inFileSpec.name` field.

g. Set the flag that says this document has a file.

Set the document's `mIsSpecified` data member to true.

```
// Delete the existing file object.
delete mFile;
```

```
// Make a new file object.
mFile = new LFile( inFileSpec );

// Get the proper file type.
OSType theFileType = 'TEXT';
if ( inFileType != fileType_Default )
  theFileType = inFileType;

// Make new file on disk.
mFile->CreateNewDataFile( 'ttxt', theFileType );

// Write out the data.
DoSave();

// Change window title to reflect the new name.
mWindow->SetDescriptor( inFileSpec.name );

// Document now has a specified file.
mIsSpecified = true;
```

`DoAESave()` handles all the details for specifying a file and setting the window title. It relies on `DoSave()` to do the actual work of saving a file.

9. Save a document.

`DoSave()` `CTextDocument.cp`

This function requires that a file already exist on disk. It's purpose is to write the data to disk. This is a very straightforward process. You can rely on the fact that the `mFile` data member connects you to a real file on disk.

To save data you must:

a. Open the data fork.

Use an LFile member function. Use the Toolbox constant `fsRdWrPerm` for read/write permission.

b. Get the data from the text view.

Declare a local `Handle` variable to receive the data. You have a pointer to the text view in `mTextView`. Use an LTextEditView member function to get the data.

c. Lock the data in memory.

You can use the StHandleLocker utility class to take care of this for you. Simply declare a local variable of that class. Specify the `Handle` you want to lock.

d. Write the data to the file.

Use an LFile member function to write the data referred to by the local `Handle` variable.

e. Close the data fork.

Once again, use an LFile member function. If you have decided to keep the file open while the document is open, you can skip this step.

f. Mark the file as clean.

Call the text view's `SetDirty()` function. Pass `false` as a parameter. Now that you have saved the file, the state of the document and the file are identical.

```
// Open the data fork.
mFile->OpenDataFork( fsRdWrPerm );

// Get the text from the text view.
Handle theTextH = mTextView->GetTextHandle();

// Lock the text handle.
StHandleLocker theLock( theTextH );

// Write the text to the file.
mFile->WriteDataFork( *theTextH, ::GetHandleSize( theTextH ) );

// Close the data fork.
mFile->CloseDataFork();
```

```
// Saving makes doc un-dirty.
mTextView->SetDirty( false );
```

You can now completely save a document. You're almost done. There are two tasks remaining: reverting a document, and letting PowerPlant know whether the document has been modified or not.

10. Revert a document.

`DoRevert()` `CTextDocument.cp`

The ability to revert a document to its previously-saved state is a user-friendly feature that should be supported in all applications that save documents. Implementing this functionality is simple. You have already performed all of the necessary tasks in other functions. There is nothing new here. You must open the file, read the data, and replace the data in the document.

The specific tasks are:

a. Open the data fork of the file.

Use an LFile member function.

b. Read the entire contents of the file and close it.

Again, use LFile member functions. Get the data in a `Handle`. Declare a local `Handle` variable to store the handle. Don't forget to close the fork when you're done, unless your following the strategy of leaving the file open while the document is open.

c. Put the data in the text view.

You have a pointer to the text view in the `mTextView` data member. Use an LTextEditView member function to set the contents of the CTextView object.

Also, because this is now a pristine view unchanged from the state of the file, set the text view's dirty flag to false. Use the text view's `SetDirty()` member function.

d. Dispose of the data.

Use the Toolbox call `DisposeHandle()` to dispose of the local `Handle`. LTextEditView makes its own copy of the data.

e. Refresh the view.

You're changing the contents of the view, so mark the view for updating. Call the view's `Refresh()` method.

```
// Open the data fork.
mFile->OpenDataFork( fsRdWrPerm );

// Read the file contents and close the file.
Handle theTextH = mFile->ReadDataFork();
mFile->CloseDataFork();

// Put the contents in the text view and clear the dirty flag.
mTextView->SetTextHandle( theTextH );
mTextView->SetDirty( false );

// Dispose of the text.
::DisposeHandle( theTextH );

// Refresh the text view.
mTextView->Refresh();
```

11. Determine if the document has been modified.

`IsModified()` `CTextDocument.cp`

From time to time PowerPlant needs to know whether a document has been modified. For example, at menu updating time, the document's modified state controls whether the Save item is enabled.

This function is very simple. Simply ask the text view if it is dirty. Store the result in `mIsModified` and return the result.

```
// Document changed if the text view is dirty.
mIsModified = mTextView->IsDirty();
return mIsModified;
```

12. Build and run the application.

When the application launches, an empty text window should appear. You can type some text into the window, then choose the **Save** item in the **File** menu. Notice that because there is no file associated

with this document, you'll see the `StandardPutFile` dialog. Specify a name and save the file. Then close the window.

Now choose **Open** from the **File** menu. The `StandardGetFile` dialog appears. Locate your file, and open it. There's your text, in all its glory. You can open other text documents as well. Give it a shot.

Make a few changes, then choose **Revert** from the **File** menu. PowerPlant displays an alert asking you to confirm the operation. Click OK, and your document reverts to the previously-saved condition.

Finally, observe the items in the **File** menu after you save a document, and again after you change the document. When the file is clean, the Save item is disabled. When the file is dirty, the Save item is enabled. Make a new window, and examine the Revert item. You cannot revert the document, it has no file.

Continue exploring until you have satisfied yourself that the application now can open, close, save, and revert documents fully and completely. When you have finished, you can quit the application.

If you'd like to continue exploring this topic, there is a lot of room for experimentation. For example, add some other objects to the window that require you to save data. Collect and save that data. Restore the objects to the correct state when opening the file.

Have a good time exploring, but don't get lost! There are only two more chapters to go. Next we talk about printing. After that comes the frosting on the cake.

14

Printing

In this chapter we discuss printing in a PowerPlant application.

Like file I/O, printing is typically the responsibility of a document. If you have not already done so, you should read "The Document Strategy" on page 434 to ensure that you understand the role of the document in a PowerPlant application. In this chapter we assume you are familiar with document concepts.

PowerPlant's approach to printing involves the collaboration of several different classes. As a result, it will help a great deal to look at the overall printing strategy before we go into class-level details. With that in mind, the topics in this chapter are:

- Printing Strategy—an overview of printing in PowerPlant.
- LPrintout—the attributes and behaviors of the fundamental printing view in PowerPlant.
- LPlaceHolder—details about this unusual view class.
- Printing in Views and Panes—printing-related behaviors in visual classes.
- The Mac OS, LPrintout, and LPlaceHolder—printing rectangles in the Mac OS and PowerPlant.
- Printing in PowerPlant—how to get your document on paper.

As usual, we'll end the chapter with a summary and code exercise.

Printing Strategy

PowerPlant creates a special view for managing printing, LPrintout. The member functions of this class provide you with all the routines you need to loop through your document's pages and print them. LPrintout mediates between your application and the printer, accommodating things like paper size, page breaks, which objects ap-

pear on which page, and so forth. In the default implementation, each pane's `DrawSelf()` function ultimately draws the pane's contents onto the printed page.

A printout contains one or more instance of a special view called a placeholder. The placeholder interacts with the printout—its superview—to determine printing dimensions based on paper size.

The placeholder begins life as an empty view with no contents. However, after you create the placeholder and before you print, you give the placeholder a single occupant. That occupant is another view—one of the views you have already encountered. It might be a single LTextEditView. It might be an LView object containing an arbitrary number of subpanes.

The critical feature here is that the placeholder itself has a single occupant, so it has one view with which it communicates. That view contains all the subviews and panes to be printed within that placeholder's boundaries.

To install the occupant into the placeholder, you temporarily move the view from its original visual hierarchy into the printing hierarchy. The placeholder takes care of putting the view back where it belongs when printing is complete.

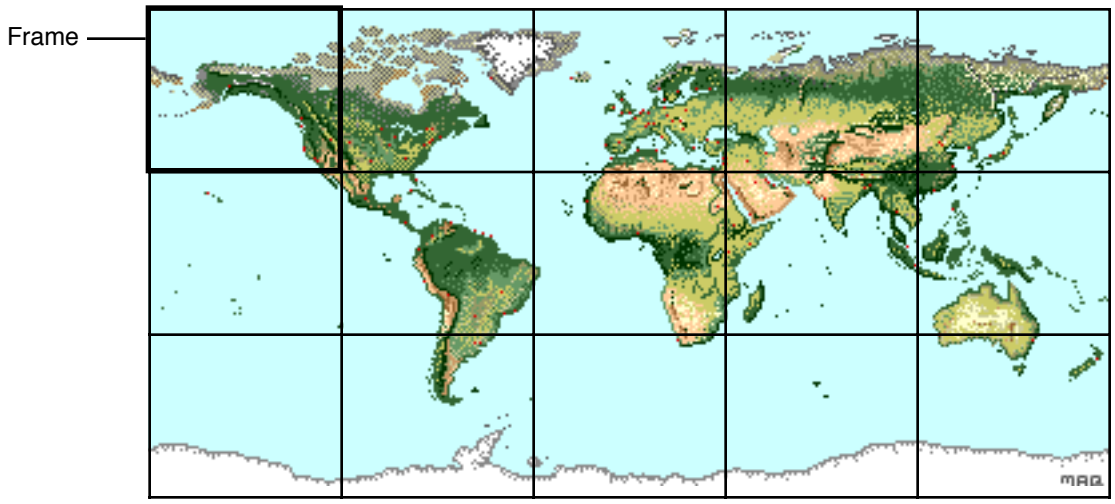
After you have installed the occupant, you're ready to print. We'll talk about the code-level details required to accomplish this task later on in this chapter. In this section, we want to concentrate on what happens when you tell the printout to print itself.

Because the printout is at the top of a visual hierarchy—in this case the printing hierarchy—it can ask all of its subpanes for information. In this context, the piece of information of greatest interest to the printout is, essentially, how big are the panes? The printout must determine how many pages there are in the document, and on which page each pane should be printed.

To accomplish this task, PowerPlant introduces the concept of a *panel*, as distinguished from a page.

Remember that a view (and LPrintout is a view) has both an image and a frame. A panel is one "framefull" of an image. Figure 14.1 illustrates the concept of a panel in relation to a frame.

Figure 14.1 Frames and panels



This image has 15 panels, extending five wide by three high. You would take that many frames to completely cover the image. In this particular example, the image is perfectly divisible by the size of the frame, but that's not necessary. PowerPlant always rounds up to ensure that the number of panels fully covers the image.

Of course, if the frame changes size, the number of panels also changes. Each view is responsible for figuring out the number of panels required to cover its entire image, using its contained views and panes in the process. Ultimately, LPrintout is the highest level view, and it keeps track of the total number of panels in its contents, and puts them on the correct page when printing.

PowerPlant contains code to count panels, measure panels, determine which panel is on which page, and so forth. Happily, you don't have to concern yourself with these details. That's the beauty of a framework. Let PowerPlant take care of figuring out low-level details like panels and pages. You can concentrate on high-level concepts like printing the document without worrying too much about counting pixels.

Now let's take a look at the actual classes that implement this strategy, and see how to use them effectively.

LPrintout

The functionality encapsulated in LPrintout gives you almost everything you need to print in PowerPlant. To support this functionality, LPrintout has several features that no other view class has. In this section we discuss

- LPrintout Characteristics—the features of LPrintout.
- LPrintout Behaviors—the functions you'll encounter while using LPrintout.

LPrintout Characteristics

LPrintout has several features necessary for its work—printing panes and views. In this section we discuss LPrintout's:

- Frame
- Data members
- Page numbering

Frame

As you know, all views have a frame. In LPrintout, the frame is the paper size in the current printer record. If the user changes printers or paper size, PowerPlant updates the LPrintout frame.

In LPrintout, the (0,0) point in local/image coordinates is the top left of the paper rectangle. As a result, all coordinates are the absolute location on the paper. This simplifies setting margins and otherwise placing panes for printing. We will revisit this topic when we discuss "Adding a placeholder" on page 485.

Do not confuse the paper size with the printable area supported by a printer. The printable area is usually smaller than the paper rectangle because most printers have mechanical limitations that prevent them from printing to the very edge of the paper.

Data members

In addition to its unusual use of the frame characteristic, LPrintout has the data members listed in Table 14.1

Table 14.1 LPrintout data members

Data Member	Stores
`mPrintRecordH`	a handle to the printer record.
`mPrinterPort`	pointer to the printer port
`mWindowPort`	pointer to a window port used by LPrintout
`mHorizPanelCount`	number of panels horizontally in printout
`mVertPanelCount`	number of panels vertically in printout
`mAttributes`	LPrintout attributes
`mForeColor`	foreground color for printing
`mBackColor`	background color for printing

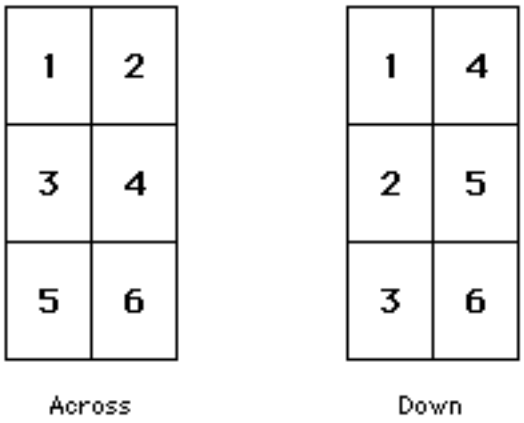
Most of this information is maintained for you automatically by PowerPlant.

Page numbering

The only attribute currently defined in PowerPlant for printing determines whether to number pages down or across in a large, multipage document.

Assume you have a document that is six pages long—two pages wide and three pages deep. Should page two be the page to the right of page one or the page below page one? Figure 14.2 illustrates the alternatives.

Figure 14.2 Page numbering in PowerPlant



By default, PowerPlant counts across, so that all pages along the top row print first. You can modify this attribute using LPrintout's attribute accessors `HasAttribute()` and `SetAttribute()`. The constant to count pages down is `printAttr_NumberDown`.

LPrintout Behaviors

LPrintout has a series of functions that implement printing in the Mac OS using standard QuickDraw.

Table 14.2 lists the LPrintout printing functions.

Table 14.2 LPrintout printing functions

Function	Purpose
`DoPrintJob()`	print the contents of the LPrintout view
`PrintPanelRange()`	open printer driver, call `PrintCopiesOfPages()`
`PrintCopiesOfPages()`	the printing loop to print each page
`GetPrintJobSpecs()`	extract information about the print job from the Toolbox print record

Function	Purpose
`CountPanels()`	determine how many panels are in a printout
`PrintPanel()`	print a specified panel

All of these functions are fully realized in PowerPlant, and most are used internally. The only one you are likely to call directly is `DoPrintJob()`. We'll discuss the circumstances under which you make this call in "Printing a Document" on page 487.

You may have noticed that there are no functions for common, printing-related tasks such as displaying the print job dialog, the page setup dialog, and so forth. Those functions are in a utility class, UPrintingMgr, discussed in "Printing Utilities" on page 493.

In summary, most of the features and behaviors in LPrintout are only of indirect interest to you. PowerPlant uses them internally. The important facts to remember are that LPrintout uses the paper size for its frame, can number pages across or down, and has the `DoPrintJob()` function to print its contents.

Now let's take a look at the placeholder.

LPlaceHolder

LPlaceHolder is a unique view class designed to assist the printing process. The purpose of a placeholder is to allow you to print a pane at a size and/or location that is different from the pane's characteristics when displayed in a window. It is a simple class with very few (but important) distinctions between itself and LView.

Like LPrintout, let's discuss

- LPlaceHolder Features
- LPlaceHolder Behaviors

LPlaceHolder Features

A placeholder has two characteristics that differentiate it from other views. It has an occupant, and the occupant has an alignment.

The occupant

There may be one and only one occupant in a placeholder. The occupant must be another pane (including most views and controls). However, the occupant cannot be an LWindow or LDialogBox. LWindow (and any class derived from LWindow) must be a top-level view and cannot reside inside LPlaceHolder.

When you install the occupant in the placeholder, the placeholder stores the occupant's original size, location, and superview. When you remove the occupant from the placeholder, or delete the placeholder, the placeholder automatically restores the occupant to its original size, location, and superview.

Alignment

While in the placeholder, the occupant has an alignment. This locates the pane inside the placeholder's dimensions. PowerPlant uses the Mac OS Toolbox `AlignmentType` values. The possible values are defined in the icons.h file in the universal headers. The possible alignments are:

`kAlignNone`	`kAlignLeft`
`kAlignVerticalCenter`	`kAlignCenterLeft`
`kAlignTop`	`kAlignTopLeft`
`kAlignBottom`	`kAlignBottomLeft`
`kAlignHorizontalCenter`	`kAlignRight`
`kAlignAbsoluteCenter`	`kAlignCenterRight`
`kAlignCenterTop`	`kAlignTopRight`
`kAlignCenterBottom`	`kAlignBottomRight`

If you specify no alignment, the frame of the occupant pane—typically a view—is resized to fit the placeholder dimensions. If you don't specify a horizontal alignment, the occupant width is set to the placeholder width. Similarly, if you don't specify a vertical alignment, the occupant height is set to the placeholder height.

The "no alignment" option allows you to keep the frame of the occupant view automatically adjusted to the size of the placeholder. This resizing is very important in a typical application.

The placeholder frame controls the print area of your document. You can think of the placeholder frame as the panel size for the document. If the occupant view's frame matches the placeholder frame, then your occupant view's frame fills the printing panel. If the occupant view's *image* is less than or equal to the frame size, then you get one page. If the occupant view's image is greater than the frame size, you get multiple pages.

Subpanes within the occupant view are not resized. They remain at the same size and position relative to the top left corner of their superview.

LPlaceHolder Behaviors

Other than constructors and destructor, LPlaceHolder has two new behaviors that it adds to those it inherits from LView. LPlaceHolder also overrides two panel-related functions it inherits from LView. Table 14.3 lists all four functions.

Table 14.3 LPlaceHolder functions

Function	Purpose
`InstallOccupant()`	put pane in placeholder, preserving original size, location, and superview, resizing to fit placeholder if necessary
`RemoveOccupant()`	restore occupant to original condition
`CountPanels()`	tell occupant to count panels
`ScrollToPanel()`	tell occupant to scroll to panel

Based on the pane and alignment you specify, `InstallOccupant()` preserves the pane's original state and resizes the pane to fit the placeholder as necessary. `RemoveOccupant()` restores the

original state. The LPlaceHolder destructor calls `RemoveOccupant()`.

The panel-related functions tell the occupant to perform the requested task. They add no new functionality to these behaviors.

Printing in Views and Panes

Although we have ignored them until now, all views and panes have printing-related behaviors. This section gives you background information on these functions. With one exception, it is unlikely that you will ever override or modify any of these functions. PowerPlant printing functionality is virtually complete.

Printing-related functions in views and panes perform three tasks:

- count and scroll panels
- dispatch control to each pane to be printed
- print the pane

Printing follows the visual hierarchy. You start with a top-level view and work down to the leaf-level panes. Let's look first at the printing functions in views, and then at panes.

Table 14.4 lists the five printing-related functions in views.

Table 14.4 View printing functions

Function	purpose
`CountPanels()`	return number of panels horizontally and vertically in this view
`ScrollToPanel()`	scroll view to specified panel
`PrintPanel()`	perform necessary housekeeping, call `PrintPanelSelf()` for view, call `SuperPrintPanel()` for subpanes

Function	purpose
`SuperPrintPanel()`	superview is printing this view; perform necessary housekeeping, call `PrintPanelSelf()` for view, call `SuperPrintPanel()` for subpanes
`PrintPanelSelf()`	uses inherited LPane function—the default calls `DrawSelf()`

Table 14.5 lists the same five functions as implemented in panes.

Table 14.5 Pane printing functions

Function	purpose
`CountPanels()`	for panes, always one
`ScrollToPanel()`	panes do not scroll, pane is valid
`PrintPanel()`	perform necessary housekeeping, call `PrintPanelSelf()`
`SuperPrintPanel()`	superview printing this pane; call `PrintPanel()`
`PrintPanelSelf()`	default calls `DrawSelf()`

A pane always has one panel. Remember, a panel is a "framefull" of the image. In a pane, the image and frame are the same size.

All of the functions for counting panels and passing the printing request down through the visual hierarchy to the final panes is complete and fully realized in PowerPlant. You typically will not have to modify this process.

Of all of these functions, the only one you are likely to override is `PrintPanelSelf()`. We talk about that in "Printing a Document" on page 487.

The Mac OS, LPrintout, and LPlaceHolder

Before we get into the real work of printing a document in PowerPlant, let's take a quick look at how the Mac OS handles some basic printing rectangles, and relate that to PowerPlant's use of the same rectangles. This should help ease your conversion to PowerPlant printing. Table 14.9 lists some printing rectangles as they are expressed in both the Mac OS and PowerPlant.

Table 14.6 Mac OS and PowerPlant printing rectangles

Concept	Mac OS	PowerPlant
paper size	`TPrint.rPaper`	LPrintout frame
printable area	`TPrint.prInfo.rPage`	n/a
printing area	n/a	LPlaceHolder frame

The paper size is the size of a sheet of paper. The printable area is the area where the printer is capable of printing. The printing area is the area where you are actually printing. Typically, the printable area is a rectangle within the paper size, and the printing area is a rectangle within the printable area.

If the printing area is larger than the printable area, or offset so that part of the printing area extends outside of the printable area, part of your image will not print.

In PowerPlant, the LPlaceHolder frame controls the printing area. This should be less than or equal to the printable area. The LPlaceHolder frame should also be positioned within the LPrintout frame so that it stays on the paper.

Typically you set the position and size of the LPlaceHolder frame in Constructor. At that time you may make assumptions about the size of the paper and the printable area. At runtime, you may want to modify LPlaceHolder to adjust to the actual printable area. Check the LPlaceHolder frame against the Mac OS `TPrint.prInfo.rPage` information.

Printing in PowerPlant

Fundamentally, printing is a visual task. LPrintout is a view class. Panes and views handle most of the low-level drawing, whether on screen or to a printer.

This section has these principal topics:

- Building a Printing Hierarchy—using Constructor to set up the printing hierarchy, and doing the same on the fly.
- Printing a Document—the tasks you perform and functions you override to get your document onto paper.
- The Print Record—how PowerPlant maintains the print record, and why you might want to override it.
- Printing Utilities—PowerPlant printing utility functions.

Building a Printing Hierarchy

You can create a printing hierarchy using Constructor, or on the fly in your code. We talk about each method. Then we discuss what to do when you derive your own class from LPrintout.

Using Constructor

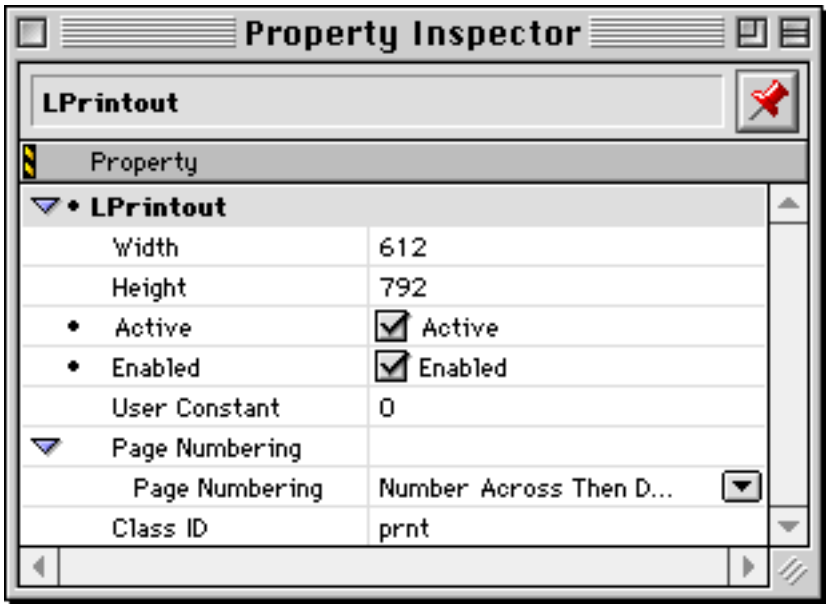
Printing uses the same PPob resource concept with which you are familiar. If you have built a PPob resource in Constructor, creating the printing hierarchy is simple. You call `LPrintout::CreatePrintout()` with the resource ID number for the PPob resource that describes the LPrintout object.

Creating a printout in Constructor is simple. While in the Constructor project window, with no resource or a PPob resource selected, choose **New Resource** (command-K) from the **Edit** menu. When you do, the dialog in Figure 14.3 appears.

Figure 14.3 Creating a new printout

Choose LPrintout as your view type. Click the create button to create the new PPob resource. Open the new PPob resource, and then the Property Inspector window for this particular printout, as shown in Figure 14.4. This is where you set the printout characteristics.

Figure 14.4 Setting printout properties with Constructor



A printout has width, height, class ID, and a user constant. The width and height aren't that important because PowerPlant resizes the printout to match the paper size of the current printer record.

Set the printout to be active and enabled. You can also set your page numbering option. See "Page numbering" on page 475.

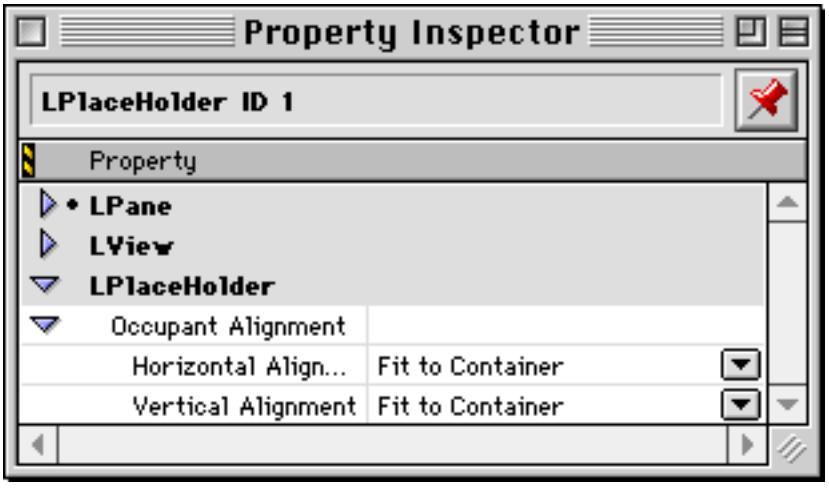
The other characteristics of panes and views are not used. Remember, if you derive your own printout class you must change the class ID to your own unique value and register the class with PowerPlant before creating any objects of that class.

See also "Register PowerPlant Classes" on page 288.

Adding a placeholder

Inside the printout view, you add one or more placeholders—objects of the LPlaceHolder class. The placeholder describes the bounds of a printable area of your document. In many cases, the printout has a single placeholder. After installing each placeholder, set the placeholder's characteristics. Figure 14.5 shows the window.

Figure 14.5 Setting LPlaceHolder properties



The placeholder has many of the features of other views.

The placeholder's position, size, and binding relative to the superview are important. The superview is the LPrintout object. Power-

Plant resizes the printout object to match the *paper size* of the printer—not the printable area. If you put the placeholder at the very top left corner of the printout—setting both top and left to zero—part of your view's image area will be truncated. Most printers cannot print to the edge of the paper.

If you want your placeholder to occupy all but a certain margin around the edge of the paper, put the top left corner of the placeholder where you want the top left pixel to appear on the paper. For example, a margin of one inch would be 72 pixels. Then set the size to provide the proper margins on the right and bottom. Finally, make sure that frame binding is on for all four sides. Then, when PowerPlant resizes the printout to match whatever the paper size is, your placeholder will resize properly and still give you the correct margin.

Scrolling is typically of no relevance to a placeholder, so the values for image size, scroll unit, scroll position, and reconcile overhang are unimportant.

Set the placeholder's ID, and make sure it is enabled and visible.

The only new characteristic of a placeholder is its alignment. If you do not specify an alignment in your code, PowerPlant uses the alignment specified in the PPob. Typically, you use no alignment. In that case, the view ultimately installed as the occupant in the placeholder resizes automatically to the dimensions of the placeholder.

You do not need to create a placeholder for every single pane and view in your visual hierarchy. You only need a single placeholder. At runtime you place a view into the placeholder as its sole occupant. That view should contain all the panes you want printed.

Creating a printout on the fly

If you wish to create a printout on the fly, you can define a new printout using the default LPrintout constructor. This sets up and initializes the printout to default values. You can then modify the printout's characteristics as necessary. Study the LPrintout source code and *PowerPlant Revealed* for details.

You will also need to create one or more LPlaceHolder views and install them inside the printout.

Deriving your own printouts

Creating your own printout class is a fairly unusual occurrence. Most of the traditional printing functionality in the Mac OS is built right into PowerPlant, thus eliminating the need to subclass from LPrintout. However, there are reasons why you might want to derive a new printout class with additional functionality.

For example, you might want to implement a different printing architecture, such as QuickDraw GX. You might want to add error control or display a custom printing status dialog. In the latter case, you are very likely to override the functions listed in Table 14.7.

Table 14.7 Commonly overridden LPrintout functions

Function	Purpose
`PrintPanelRange()`	open and close printer driver
`PrintCopiesOfPages()`	print a range of pages

If you examine the source code for these two functions you'll see that the locations where you need to add code are already mapped out for you.

Printing a Document

After you have the printing hierarchy set up, the steps you follow to print a document in PowerPlant are simple and straightforward. The details may become very complex depending upon the nature of your documents. As always, implementation details are independent of PowerPlant.

This section discusses the general steps you must follow, and gives suggestions for typical ways in which you might modify the standard approach to printing.

In this section we talk about how to:

- Print from the Finder
- Print a Document
- Print a Pane

Print from the Finder

Printing from the Finder is a much-ignored feature of the Macintosh human interface. In a well-designed Mac OS application, the user can select document icons on the desktop and choose **Print** from the Finder's **File** menu.

In response, your application prints the selected documents. Your application may or may not be running at the time. If your application is running, the document or documents chosen may not be open.

When the user prints from the Finder, your application receives a print document event. In PowerPlant, the application object's `DoAEOpenOrPrintDoc()` function handles it. The default PowerPlant function calls the application's `PrintDocument()` function.

In both PowerPlant application classes—LApplication and LDocApplication—this function is empty. You must implement this function in your derived application class if you support printing from the Finder.

The steps to follow in your application's `PrintDocument()` function are clearly defined. They are:

1. Open the document if it is not already open. You receive a Mac OS FSSpec record for the document to be printed. If the document is not already opened, it should never be displayed on the monitor, just printed.
2. Print the document. In a well-factored application, this usually means calling the document's printing behavior.
3. Close the document if it was not already open.

You do not need to worry about quitting your application. If the user is printing from the Finder, the Finder takes care of that for you by sending the necessary quit application Apple event.

Finally, if you do not support printing from the Finder, you can leave `PrintDocument()` empty. It is not used as part of the normal document printing from within an application.

Print a Document

When the user chooses the **Print** item in your application's **File** menu, your application prints the document. Responsibility for handling this command will rest with whatever commander class identifies and responds to the printing command in its `ObeyCommand()` function. For example, you might want to have a derived window class respond to this command to print its contents.

However, it is more typical to have a document respond to this command. You may recall from the previous chapter that LDocument has both `ObeyCommand()` and `DoPrint()` functions.

The default `LDocument::ObeyCommand()` function handles the Print command completely. The function ensures that there is a print record. It displays the standard print dialog. It sends an apple event for recording purposes. Then it calls the `DoPrint()` function.

NOTE: The default `ObeyCommand()` does nothing if there is no print record. You may want to override this behavior to tell the user why printing failed.

However, the default `LDocument::DoPrint()` function is empty. In your derived document class, you implement this function. The tasks to accomplish in `DoPrint()` are:

1. Create the printout view.
2. Set the print record for the printout view.
3. Install the occupant view in the placeholder.
4. Call the printout view's `PrintJob()` function.
5. Delete the printout view.

Listing 14.1 contains sample code for a typical `DoPrint()` function.

Listing 14.1 An example DoPrint() function

```
CMyDoc::DoPrint()
{
  // make printout view
  LPrintout* thePrintout = LPrintout::CreatePrintout(prtoID);

  // set print record
  thePrintout->SetPrintRecord(mPrintRecordH);

  // install occupant in placeholder
  LPlaceHolder* thePlace = dyanmic_cast<LPlaceHolder *>
                    (thePrintout->FindPaneByID(placeID));
  thePlace->InstallOccupant(mView, atNone);

  // print it - sweet and simple
  thePrintout->DoPrintJob();

  // clean up
  delete thePrintout;
}
```

In this example, the code first creates the LPrintout view, just as it would any other view. It sets the printout view's print record. It then gets a pointer to the LPlaceHolder pane inside the printout view.

The CMyDoc class has a data member, `mView`, that stores the view representing the document. The code installs that view inside the placeholder. It then calls the printout's `DoPrintJob()` function to print the contents of the view. Finally, it deletes the printout.

TIP: The view you place as occupant inside the placeholder cannot be an LWindow. LWindow must be a top-level view. If you want a single view encompassing the contents of a window, put a simple LView object inside the window, then put the window contents inside the enclosing LView object.

Print a Pane

As the printing process in PowerPlant progresses, the ultimate responsibility for drawing an individual pane falls on that pane's `PrintPanelSelf()` function. In this section, we use the term "pane" in its most general sense to include all panes—including views. The default implementation of `PrintPanelSelf()` simply calls the pane's `DrawSelf()` function.

Override `PrintPanelSelf()` if you want a pane to print differently than the way it draws on screen. There are several common reasons why you might want to do this.

If your pane erases and then draws, you might want to eliminate erasing. Erasing is a major cause of slow printing. You don't have to erase a blank sheet of paper. For example, a window view erases itself before drawing if the `EraseOnUpdate` attribute is set.

You may want to add items to the printed document that do not appear in a view. You might want to add page numbers, or a header or footer to a document. Conversely, you may not want to print certain items that appear on screen but that are unimportant in the printed document.

If your pane draws offscreen and uses `CopyBits()` to draw, you may want to replace that behavior for printing. Text that is blitted to the printer port comes out jaggy. Text that is drawn directly to the printer port comes out smooth.

If a pane crosses a page boundary, you might want to ensure that the break occurs at a reasonable spot. Again using text as an example, the page boundary might slice right through the middle of a line of text. That text should appear on the following page, rather than have the top half of the line appear at the bottom of one page, and the bottom half of the line appear at the top of the next page.

Resolving these sorts of boundary problems is non-trivial, but they are beyond the scope of PowerPlant. An application framework does the general work. You extend that framework to meet your application's individual requirements. Along the way, you may encounter a need to use certain utility functions in PowerPlant.

The Print Record

PowerPlant keeps two different kinds of print records. There is a default or "global" print record. In addition, every PowerPlant document has an `mPrintRecordH` data member to store a `THPrint` handle to its own Mac OS `PrintRecord`.

The default Print and Print One command handlers in LDocument use the document's print record to control the printing process.

However, the document does not handle the Page Setup command itself. In the default implementation in PowerPlant, that task resides in LDocApplication. LDocApplication responds to the Page Setup command by changing the values in the default or global print record, not the print record in any document. This is as it should be. At the application level (that is, when no documents are open), you should modify a default print record.

When a document is open, a good application should modify the document's print record. This behavior does not exist in LDocument or LSingleDoc. To implement this behavior, you must override the `LDocument::ObeyCommand()` function to identify and handle the Page Setup command. The code might look like this:

Listing 14.2 Modifying a document's print record

```
case cmd_PageSetup:
  UDesktop::Deactivate();
  if (mPrintRecordH == nil)
  {  // get default print record
    THPrint defaultPrintRecord =
                  UPrintingMgr::GetDefaultPrintRecord();
    // make a copy
    mPrintRecordH = defaultPrintRecord;
    ThrowIfOSErr_( ::HandToHand(&(reinterpret_cast<Handle>
                        (mPrintRecordH)));
  }
  UPrintingMgr::AskPageSetup(mPrintRecordH);
  UDesktop::Activate();
  break;
```

You would save the print record with the document, and restore it when opening a file. We discussed file I/O in the previous chapter.

TIP: If a document has no print record, `LDocument::ObeyCommand()` creates a new print record by calling `UPrintingManager::CreatePrintRecord()`. You might want to override this behavior and call `UPrintingMangager::GetDefaultPrintRecord()`, and make a copy of the default record. Changes made when no windows are open are then used for new windows by default.

Printing Utilities

Should you find yourself required to deal with the Mac OS Printing Manager, you should use the functions in UPrintingMgr. This class is a wrapper for the most common Printing Manager calls you are likely to make. Table 14.8 lists all the available functions.

Table 14.8 UPrintingMgr functions

Function	Purpose
`OpenPrinter()`	open the current printer driver
`ClosePrinter()`	close the current printer driver
`CreatePrint Record()`	create a TPrint record with default settings and return a handle to it
`GetDefaultPrint Record()`	return handle to the default settings maintained by this class
`ValidatePrint Record()`	check if a TPrint record is valid for the current printer driver, change if necessary
`AskPageSetup()`	display the standard page setup dialog
`AskPrintJob()`	display the standard print job dialog

Each of these functions is a static function, so they are always available. Consult *PowerPlant Revealed* and the source code for details.

Summary

In this chapter you learned how several PowerPlant classes work together to implement printing.

LPrintout is responsible for managing most of the printing tasks. Its dimensions match the paper size for the printer, it counts panels, converts panel number to page number, has the main printing loop, and so forth.

LPlaceHolder keeps track of its occupant view. The placeholder dimensions describe the printing area on each page.

You create a printing hierarchy consisting of a printout and one or more placeholders.

In your document class you write the `DoPrint()` function. You identify the view that contains all the panes you want printed. You install that view as the placeholder's occupant, and print.

If necessary, you override the `PrintPanelSelf()` functions in various pane and view classes to customize printing behavior.

To ease your conversion to PowerPlant printing, Table 14.9 lists some printing concepts as they are expressed in both the Mac OS and PowerPlant.

Table 14.9 Mac OS and PowerPlant printing data

Concept	Mac OS	PowerPlant
paper size	`TPrint.rPaper`	LPrintout frame
printable area	`TPrint.prInfo.rPage`	n/a
printing area	n/a	LPlaceHolder frame

The paper size is the size of a sheet of paper. The printable area is the area where the printer is capable of printing. The printing area is the area where you are actually printing.

If the printing area is larger than the printable area, part of your image will not print.

In PowerPlant, the LPlaceHolder frame controls the printing area. This should be less than or equal to the printable area. Typically you set the dimensions of LPlaceHolder in Constructor. However, if you want to modify LPlaceHolder at runtime to match the printable area, or to ensure that the printing area is smaller than the printable area, check the LPlaceHolder frame against the Mac OS `TPRint.prInfo.rPage` information.

Code Exercise

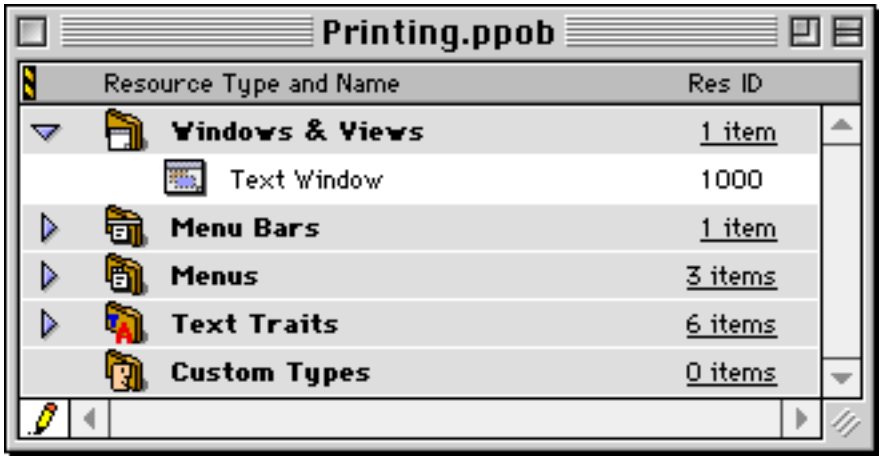
In this exercise you add printing functionality to the same application you worked with in the previous chapter. First you create the interface and then you write the code necessary to implement printing.

The Interface

The text window remains intact and unchanged from the previous chapter. If you'd like to review the text window components, refer to that exercise. To implement printing, you need to add two elements to the interface, a printout and a placeholder.

Open the `Printing.ppob` project file in Constructor. It should look like Figure 14.6. In the start code there is already one PPob resource, the one for the text window and its contents. In this section you add another PPob resource for the printout.

Figure 14.6 The Printing.ppob file from the start code



1. **Create an LPrintout view.**

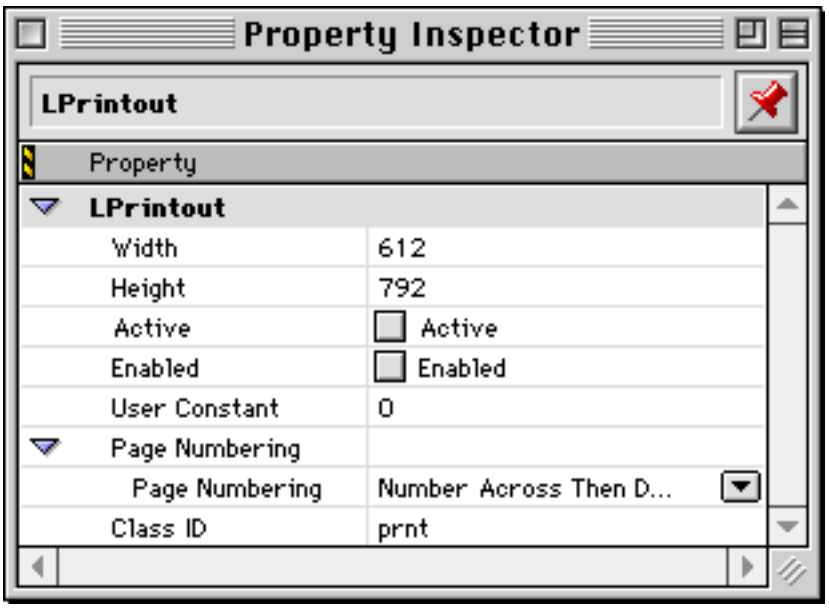
   With no items or a PPob resource selected in the Constructor project window, choose **New Resource** (command K) from the **Edit** menu. When you do, the Create New Resource dialog appears, as shown in Figure 14.7.

Figure 14.7 Creating a new PPob resource

Set the resource type to Layout. Set the view kind to LPrintout. Set the resource name. Set the resource ID to 1100. Then click the Create button.

A new entry for this PPob resource appears in the Constructor project window. Double-click the entry, and a large window opens containing the new text printout. Double click the printout view to see its characteristics, as shown in Figure 14.8.

Figure 14.8 LPrintout properties

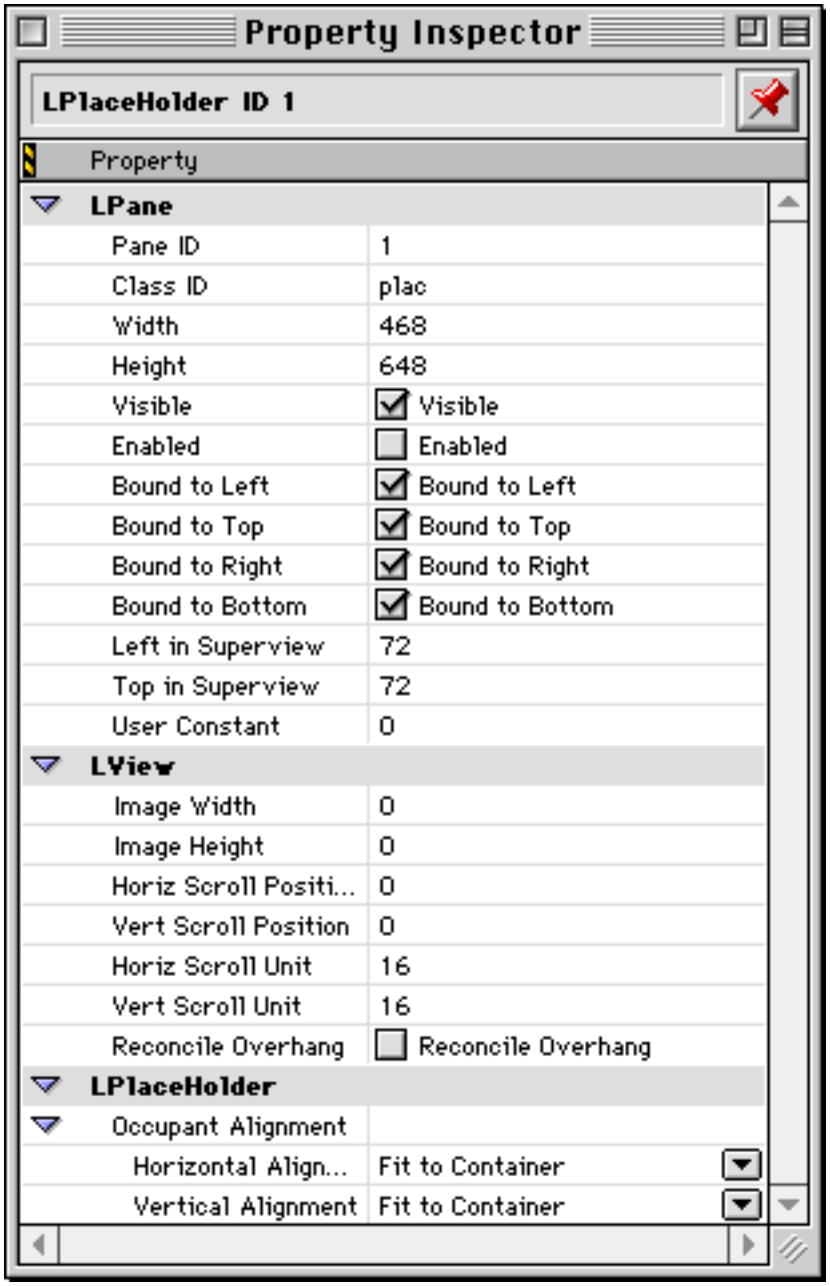


The default width and height are 8.5″ x 11″ at 72 dpi. You can turn off the Active and Enabled check boxes. Close the LPrintout characteristics window when you're through. Leave the LPrintout window open so you can add a placeholder to the printout.

2. **Create an LPlaceHolder view.**

Open the Catalog window. Drag an LPlaceHolder object and drop it onto the LPrintout view. Double-click the new placeholder to set its characteristics, as shown in Figure 14.9.

Figure 14.9 LPlaceHolder properties



The location and size of the placeholder allow for a 72-pixel (one inch) margin between the printout and the placeholder. The placeholder is bound on all sides to the printout. Set the pane ID to 1.

Make sure the placeholder is visible, but it can be disabled. Set the other characteristics accordingly. Make sure you set both the horizontal and vertical alignment to none.

There is no need to set the view hierarchy. There is only one item in the view, the placeholder. It is automatically contained in the LPrintout view.

That's it. You have just completed the printing interface for this application. Save your work and quit Constructor.

Implementing Printing

In the previous exercise you implemented opening and saving a document. In this section you implement printing a document.

Like the previous exercise, the steps take a top-down approach. That is, you start at the application level and implement the application functions necessary to support printing a document. That means supporting printing from the Finder.

Then you add printing to a document class derived from LSingleDoc. This is where the real work takes place. Happily, PowerPlant does most of the work. Let's see how.

3. Implement printing from the Finder.

`PrintDocument()` `CPrintingApp.cp`

Recall from earlier in this chapter that PowerPlant calls this function in response to a print document Apple event received from the Finder. Such an event is generated when the user selects your document on the Desktop, and chooses **Print** from the Finder's **File** menu, or drops your document on a desktop printer icon. This function is *not* called when the user chooses **Print** from your application's **File** menu.

Recall also that in the PowerPlant document strategy, a document is responsible for printing itself.

Therefore, there are two steps you must take to implement printing from the Finder.

a. Create a document.

You did this in the previous exercise. This function receives a file specification. Call the document constructor. Pass in the supercommander and the file specification.

b. Tell the document to print itself.

Call the document's `DoPrint()` function.

```
// Create a new document using the file spec.
CTextDocument *theDocument = new CTextDocument(this, inMacFSSpec);
Assert_( theDocument != nil );

// Tell the document to print.
theDocument->DoPrint();
```

That's it. You have just implemented printing from the Finder. Save your work and close the file. Of course, you have just delegated printing responsibility to the document. You implement that functionality in the next step.

TIP: This code might be a little too simple. For example, what happens if the user selects a file in the Finder that is already open in your application? This might have repercussions in your application, or it might not. Keep it in mind.

4. Print a document.

`DoPrint()` `CTextDocument.cp`

PowerPlant calls this function whenever the user chooses the **Print** item in the **File** menu. Recall that this document class inherits from LSingleDoc, and ultimately from LDocument. In LDocument, the `DoPrint()` function is empty. In your subclass (in this case CTextDocument), you must supply the necessary functionality.

In this function the document should print itself. There are several steps you must go through to accomplish this task. They are:

a. Create the printout view.

Call the LPrintout class creator function. The declared constant for the PPob resource is `rPPob_TextPrintout`. In the process of creating the LPrintout, that object is assigned the default print record by a call to `UPrintingMgr::GetDefaultPrint-Record()`.

b. Set the print record for the printout.

You want to print the document with the document's own printing options, not the default printing record. The document has an `mPrintRecordH` member. However, this data member may be nil. If the document has a print record, set the printout's print record to match the document. Call the printout's `SetPrint-Record()` function.

c. Get the placeholder.

The declared constant is `kTextPlaceholder` for the placeholder pane ID.

d. Install the text view in the placeholder.

Call the placeholder's `InstallOccupant()` function. Use no alignment. Remember, this makes the text view resize to fill the placeholder completely, which is just what you want to happen in this case.

e. Adjust the size of the frame.

Call `SetPrintFrameSize()`. This function is provided for you. This task is specific to this application, and not to printing in general. However, it does illustrate how you can adjust the printed frame to allow for aesthetic concerns.

Here's the problem. Remember, that the text view frame has just been resized to fill the placeholder. It is unlikely that an integral number of lines of text will just fit perfectly in the frame. As a result, the top part of a line of text may appear at the bottom of one page, and the bottom part of that same line of text would appear

at the top of the next page. That makes the document difficult to read, to say the least.

The `SetPrintFrameSize()` function adjusts the bottom of the text view frame so that an integral number of lines just fits in the frame. As a result, the printed document looks good.

When you implement printing in your own application, you will run into similar concerns relating to page and object boundaries.

f. Print the document.

Ah, here's the critical step. PowerPlant does all the work. Call the printout's `DoPrintJob()` function. You're done.

g. Delete the printout.

When you're through, delete the printout object. When the placeholder is destroyed, it returns the text view object to its original condition in the scrolling view in the text window.

Here's the solution code that accomplishes all these tasks.

```
// Create the printout.
LPrintout *thePrintout =
                LPrintout::CreatePrintout( rPPob_TextPrintout );
Assert_( thePrintout != nil );

// Set the print record.
if(mPrintRecordH)
{
  thePrintout->SetPrintRecord( mPrintRecordH );
}

// Get the text placeholder.
LPlaceHolder *thePlaceholder;
thePlaceholder = dynamic_cast<LPlaceHolder*>
                 (thePrintout->FindPaneByID( kTextPlaceholder ));
Assert_( thePlaceholder != nil );

// Install the text view in the placeholder.
thePlaceholder->InstallOccupant( mTextView, atNone );

// Set the frame size.
```

```
SetPrintFrameSize();

// Print.
thePrintout->DoPrintJob();

// Delete the printout
delete thePrintout;
```

Well done! That is all you have to do to implement printing. Save your work and close the file. Let's see how it works.

5. **Build and run the application.**

   When the project builds correctly and you run the application, a familiar text window appears. Enter some text, or open a text document, and print it. It should print fine. If you have more than one page of text, lines should break evenly across page boundaries.

   Admittedly, this is a simple case of printing, but it does everything you need to do to print a document in PowerPlant. There is plenty of room for experimentation and enhancement, however.

   For example, you still cannot print from the Finder. If you try, TeachText launches! This application does not have a custom creator type or a BNDL resource. Correct this problem, and try Finder printing. While in the Finder, select the icon for a document created by your application, then choose **Print** in the Finder's **File** menu. If you have a desktop printer icon, drop a document on your printer icon. Either way, your application should launch and print the document.

   The human interface guidelines say that when printing from the Finder, if the document is not already open you should not display a window. You should just print the document. Make your application follow that guideline.

   Remember the page-numbering option in PowerPlant to count pages down or across. Experiment with that setting on a large and wide document, and see what happens. Give the user the option of choosing which way to go.

   Create a printout with two or more placeholders—for example, a header or footer in a text document. Play with the margins and with alignment settings. Implement even and odd printing. The possibil-

ities are endless. As always, have a good time exploring. And when you're through, we can move on to the final chapter.

Congratulations! Fourteen down and one to go. In the next chapter we take on periodicals and attachments. These are two of the coolest features in all of PowerPlant.

15

Periodicals and Attachments

You have come a long way. This is the last chapter of the *PowerPlant Book*. At the end of this chapter we'll look back at everything we have covered so far, and discuss briefly where you should go from here.

In this chapter we close out the core elements of PowerPlant with a discussion of two important parts of the PowerPlant application framework:

- Periodicals—objects that receive time at regular intervals, either every time through the event loop or at idle time.
- Attachments—objects that modify the appearance or behavior of other objects at well-defined moments.

We saved these two topics for last for good reason.

Periodicals are very easy to implement, and can serve for any kind of time-dependent function. They aren't restricted to particular application tasks like file I/O or menu handling. Periodicals are unbounded in terms of utility. Therefore it helps to have a good understanding of PowerPlant before discussing periodicals.

That is even more true for attachments. Attachments are perhaps the most astonishingly powerful and simple concept in all of PowerPlant. They are the epitome of elegance in design and implementation. You are really going to like attachments.

Periodicals

In this section we discuss LPeriodical, and how PowerPlant implements repetitive tasks. This is a very cool and simple feature of PowerPlant. The topics are:

- What Is a Periodical—LPeriodical and its descendants.
- Periodical Characteristics—the features of a periodical.
- Working With Periodicals—how to use periodicals in your application.

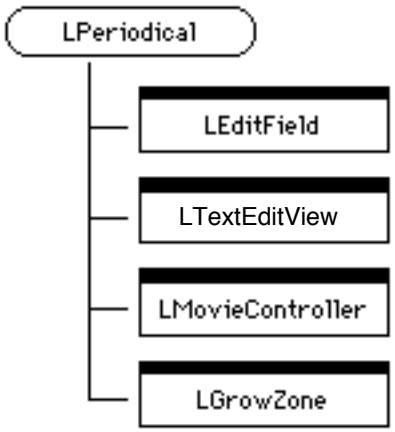
Periodicals are very easy to implement.

What Is a Periodical

In PowerPlant terminology, an object that receives time on a regular basis is a periodical. LPeriodical is an abstract base class. It inherits from no other class. Because it is an abstract class, you cannot instantiate a pure LPeriodical object. You must derive a new class that inherits from LPeriodical.

In PowerPlant, LPeriodical is used exclusively as a mix-in class. Whenever an object needs to get time repeatedly, it inherits from LPeriodical. Figure 15.1 illustrates the PowerPlant classes that inherit from LPeriodical.

Figure 15.1 LPeriodical in PowerPlant



The subclasses shown in Figure 15.1 belong to various class hierarchies in PowerPlant. Except for LGrowZone, each also inherits from other classes besides LPeriodical.

Because it is so loosely tied to the rest of PowerPlant, you can use LPeriodical separately in non-PowerPlant projects if you wish.

The only other class you need to use is LArray. LPeriodical maintains lists of objects that need regular attention. Those lists reflect the way that you use LPeriodical in PowerPlant.

Periodical Characteristics

PowerPlant deals with two kinds of periodicals, repeaters and idlers. A repeater gets time after every event. An idler gets time at every idle event.

LPeriodical has two data members, `sRepeaterQ` and `sIdlerQ`. Each of these data members is a pointer to an LArray object. Each represents a queue of objects, repeaters and idlers respectively.

Both of these data members are static, so they are class variables. There is one and only one instance of each of these variables shared by all periodical objects. Because they are static data members, any application that has periodicals has one list of repeaters, and one list of idlers.

Working With Periodicals

This section covers everything you need to work with periodicals. After looking at the member functions, we examine

- Repeaters
- Idlers
- Spending time

Table 15.1 lists the LPeriodical member functions.

Table 15.1 LPeriodical functions

Function	Purpose
`StartRepeating()`	add object to repeater list
`StopRepeating()`	remove object from repeater list
`DevoteTimeToRepeaters()`	walk through all repeaters, call `SpendTime()` for each
`StartIdling()`	add object to idler list
`StopIdling()`	remove object from idler list
`DevoteTimeToIdlers()`	walk through all idlers, call `SpendTime()` for each
`SpendTime()`	perform a periodical task

The LPeriodical destructor also deserves mention. It removes the periodical from either or both the repeater and idler queues.

Typically, the only function you override is `SpendTime()`. It is a pure virtual function and must be overridden in any subclass of LPeriodical. To understand these functions, let's look at how PowerPlant gives time to periodicals.

Repeaters

PowerPlant calls `LPeriodical::DevoteTimeToRepeaters()` every time through the main event loop. You can find this code in `LApplication::ProcessNextEvent()`. The `DevoteTimeToRepeaters()` function walks through the list of repeaters and calls each repeater's `SpendTime()` function.

To add an object to the repeater list, simply call that periodical's `StartRepeating()` function. In response, the object is added to the repeater queue. When control returns to the main event loop, the object's `SpendTime()` function will be called.

NOTE: The `SpendTime()` function will be called on the same pass through the main event loop in which the repeater is added to the repeater queue.

PowerPlant calls the `SpendTime()` function for every repeater on each pass through the event loop. PowerPlant does not call one repeater on one pass through the event loop, and another repeater on another pass.

To remove a periodical from the repeater queue, call `StopRepeating()`. You do not need to make this call if you destroy the object. The destructor does that for you.

Idlers

When PowerPlant receives an idle event or a mouse-moved event, it calls `UseIdleTime()`, which in turn calls `LPeriodical::DevoteTimeToIdlers()`. `DevoteTimeToIdlers()` walks through the list of idlers and calls each idler's `SpendTime()` function.

To add an object to the idler queue, simply call that periodical's `StartIdling()` function. In response, the object is added to the idler queue. The next time there is an idle event or a mouse-moved event, PowerPlant calls the object's `SpendTime()` function.

This is the same `SpendTime()` function called when the object is a repeater. PowerPlant calls the `SpendTime()` function for every idler for each idle or mouse-moved event.

To remove a periodical from the idler queue, call `StopIdling()`. You do not need to make this call if you destroy the object. The destructor does that for you.

TIP: A periodical can safely delete itself. The queue's for idlers and repeaters (LArray objects) are safe against insertions or removals while an iterator is traversing the list. See "Arrays" on page 547.

Spending time

A single periodical can be on both the repeater and idler queues simultaneously. You can put the same periodical on either queue at any time. The two queues are fully independent.

Membership in either queue means that the periodical's `SpendTime()` function is called. In LPeriodical, this is a pure virtual function. You must override this function in the derived class.

The SpendTime() function can do just about anything you want. You can maintain a progress bar, run a simple animation, blink a cursor, and so on. The PowerPlant classes that inherit from LPeriodical give a hint at the flexibility of this system.

LTextEditView and LEditField inherit from LPeriodical so that they can blink the text cursor. An object of either class works the same way. Each is an idler. In the `BeTarget()` function, the object installs itself in the idler queue by calling `StartIdling()`. In the `DontBeTarget()` function, each calls `StopIdling()`. In the `SpendTime()` function, each calls `::TEIdle()`. As a result, while the object is the target object, the text cursor blinks. Very simple.

LMovieController is a periodical so that it can handle QuickTime movies properly. When a movie is present, the controller should receive every event so that the Toolbox can handle movie-related events. To support this feature of the Mac OS, LMovieController is a repeater. The LMovieController constructor calls `StartRepeating()` to put the controller in the repeater queue. As long as the movie controller exists, its `SpendTime()` function is called from the event loop for each event. In `SpendTime()`, the controller receives the event and calls `::MCIsPlayerEvent()` for processing. When the movie controller is destroyed, it removes itself from the repeater queue.

LGrowZone is also a repeater. The LGrowZone constructor calls `StartRepeating()` to install itself in the repeater queue. The `SpendTime()` function implements part of the PowerPlant memory strategy we discussed in "Setup Memory Management" on page 283. If the memory reserve has been released, `SpendTime()` at-

tempts to restore the reserve. This function also warns the user of memory problems if necessary.

These four examples give you a taste of the power and flexibility of the PowerPlant periodical design. Using the identical mechanism, PowerPlant implements cursor updating, event processing, and memory management. Not bad.

Your use of LPeriodical is limited only by your imagination. If you have a situation where an object needs time repeatedly, make it a periodical. Design the `SpendTime()` function to perform the necessary tasks. Install the object in the appropriate queue—either repeater or idler—at the appropriate times, and remove it from the queue when finished.

Remember that your periodical object is not required to respond every time `SpendTime()` is called. Your object can keep track of the passage of time, and only do something if a required interval has passed. For example, you might want to update a timer every minute. Your timer object might get called several thousand times during that minute, but only act when a full minute has passed.

The periodical mechanism does not guarantee that your object will receive time within a certain time limit. The mechanism depends on the main event loop. Repeaters will get time on every pass through the event loop. If a single event requires a lot of time to process, your repeater must wait. Idlers get time at idle events. If no idle event is forthcoming, idlers get no time.

In actual practice, the event loop typically cycles several times a second. Intervals between idle events are usually very short. However, if your object is extremely fussy about receiving time at precise intervals, you will have to implement some other mechanism to ensure that your object gets called—probably a Time Manager task.

NOTE: The StDialogHandler class implements its own event loop. That loop uses the same design as the main event loop, so repeaters and idlers are called while a modal dialog based on StDialogHandler is active.

Attachments

An attachment is an object that—typically—alters the runtime behavior of another object. It is connected (attached) to the affected object. We refer to these two objects as the attachment and the host.

The attachment mechanism is very general—and very powerful. Exactly what an attachment is, and how to use one, becomes clear as we discuss:

- What Is an Attachment—classes that are attachable as well as the attachment classes.
- Attachment Strategy—how PowerPlant implements the attachment design pattern.
- Attachment Characteristics—the attributes and functions of attachments.
- Working With Attachments—everything you need to know to implement attachments.
- Specific PowerPlant Attachments—details about the attachment classes provided for you in PowerPlant.

What Is an Attachment

There are two parts to the attachment mechanism in PowerPlant: the objects to which you connect the attachments, and the attachments themselves.

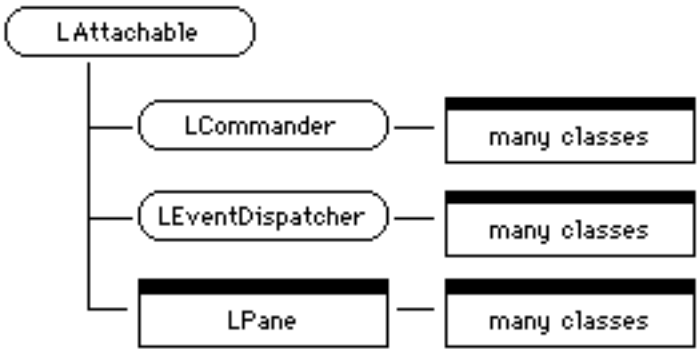
Objects to which you can connect an attachment are said to be "attachable." Be careful of the terminology here. The term "attachable" in normal usage implies that something is capable of being attached to something else. In PowerPlant, we use the converse meaning. Saying that an object is attachable means something can be attached to it.

An object to which an attachment is connected is called a host.

There are two corresponding base classes, LAttachable and LAttachment. Let's look at LAttachable first, and then at LAttachment.

Figure 15.2 illustrates the PowerPlant classes that are attachable—that is, to which you can hook an attachment.

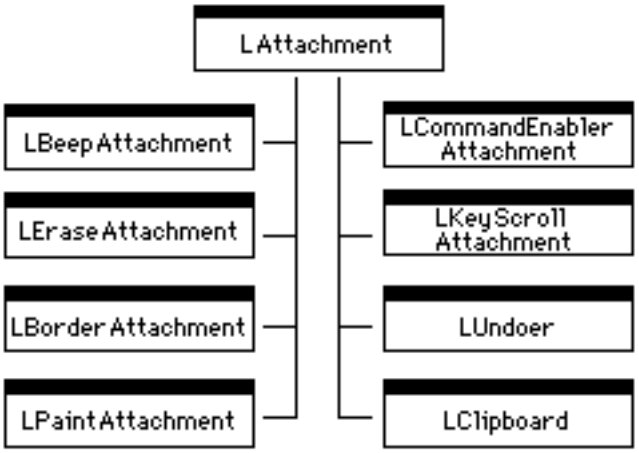
Figure 15.2 LAttachable class hierarchy



Stop for a moment and consider the import of Figure 15.2. Every commander class can have attachments—windows, applications, and so on. Event dispatchers can have attachments—applications and StDialogHandler. Every pane class—panes, views, and controls—can have attachments. In other words, every visual, command, and event-related element in PowerPlant can have attachments.

LAttachment describes the features of a generic attachment. PowerPlant also includes several attachment classes ready for your use. Figure 15.3 lists the PowerPlant attachment classes.

Figure 15.3 LAttachment class hierarchy



If you look at the list of subclasses that derive from LAttachment, you begin to get clues about the usefulness of attachments. PowerPlant uses attachments for modifying the appearance of a pane, for modifying the behavior of a pane, to support commands and keystrokes, the undo mechanism, and the clipboard. You can use attachments for other purposes as well. That's a lot of utility for one design pattern.

TIP: For programmers familiar with MacApp, PowerPlant attachments supply the functionality of the MacApp adorner and behavior classes.

We'll discuss the individual attachment classes in "Specific PowerPlant Attachments" on page 522. For now, let's get an overview of how attachments work, and then look at the features of attachments in general.

Attachment Strategy

The PowerPlant approach to attachments is similar to the periodical mechanism with which you are already familiar. It is a very simple strategy.

As you know, there are attachments and hosts. These objects are closely connected.

Each host maintains a list of its own attachments. As a result, an object may have several attachments.The host can add or remove attachments from the list, so you can modify the list of attachments at runtime.

At certain well-defined moments, PowerPlant tells the host to walk through the list of attachments. We'll discuss the precise moments in "When and how PowerPlant calls attachments" on page 518. PowerPlant sends a specific message that identifies the task that the host is about to undertake. PowerPlant also sends any necessary data that might be required to fulfill the task. This message is sent *before* the host performs its principal, underlying task.

In response, the host tells each attachment in the list to do whatever it is the attachment does. The attachment determines if the message is one to which it should respond. If the attachment is designed to respond to the message, the attachment executes.

The attachment also returns a Boolean value that tells PowerPlant whether the host object should still execute the original task.

Using this mechanism you can modify the appearance and/or behavior of any attachable object at runtime, without modifying the underlying object. You simply add or remove attachments.

That's all there is to it. The attachment mechanism is like a blend of the periodical and messaging systems. At certain well defined moments, a message is sent to each attachment. If the attachment recognizes the message, it responds and performs its task.

To implement this strategy, both attachable objects and attachments have data members and functions. Let's see what they are as we examine attachment characteristics.

Attachment Characteristics

In this section we examine both:

- Features of attachable objects
- Features of attachments

We look at LAttachable first, because it is an extremely simple class that implements the whole strategy.

Features of attachable objects

LAttachable has one data member, `mAttachments`. This is a pointer to an LArray of attachment objects.

Table 15.2 lists all the LAttachable functions, except for the constructor and destructor.

Table 15.2 LAttachable functions

Function	Purpose
`AddAttachment()`	add an attachment to the list
`RemoveAttachment()`	remove an attachment from the list
`RemoveAllAttachments()`	remove all attachments from the list
`ExecuteAttachments()`	walk the list, call each attachment's `Execute()` function

The first three functions modify the contents of the attachment list. The class destructor calls `RemoveAllAttachments()` when destroying an attachable object. You can use it directly yourself if you want to remove all attachments for any reason.

The `ExecuteAttachments()` function is the dispatch mechanism for the host. This is the function that PowerPlant calls when it wants attachments to do their work.

Typically you will never override any of these functions. LAttachable is a complete, fully-realized class that provides all the functionality you are likely to need.

Features of attachments

LAttachment is almost as simple as LAttachable. It has three data members, listed in Table 15.3.

Table 15.3 LAttachment data members

Data Type	Member	Description
`LAttachable*`	`mOwnerHost`	pointer to the host attachable object
`MessageT`	`mMessage`	message to which the attachment responds
`Boolean`	`mExecuteHost`	whether the host object should also execute

Each attachment is designed to respond to a particular message. That value is kept in the `mMessage` member. We'll discuss the possible messages in "Working With Attachments" on page 518.

The class declares eight functions. Six of these functions are simple accessors for the three data members. There are two functions designed to implement attachment functionality, listed in Table 15.4.

Table 15.4 LAttachment functions

Function	Purpose
`Execute()`	if message is right, call `ExecuteSelf()`
`ExecuteSelf()`	perform the attachment task

The `Execute()` function does the necessary testing. If the message received is the message for which the attachment is designed, it calls `ExecuteSelf()`. If the attachment executes, `Execute()` returns the value of `mExecuteHost`. Otherwise it returns `true`. PowerPlant uses this value to decide whether the host object should also perform the task in question.

The only function you typically override is `ExecuteSelf()`. In fact, in LAttachment this is an empty function. You can study the PowerPlant attachment classes like LPaintAttachment to see how they implement `ExecuteSelf()`.

Working With Attachments

In this section we discuss the code-level details you need to implement attachments in your PowerPlant application. We discuss:

- When and how PowerPlant calls attachments
- Creating your own attachments
- When to use attachments
- Uses for attachments

When and how PowerPlant calls attachments

PowerPlant asks attachments to execute before:

- any event dispatch
- any commander responds to a command
- any commander handles a keystroke
- any commander updates a menu
- any pane responds to a click
- any pane draws
- any pane prints
- any pane adjusts the cursor

At these moments, PowerPlant calls a host's `ExecuteAttachments()` function. It passes two parameters: a specific message, and a pointer to additional data.

Table 15.5 summarizes each call to `ExecuteAttachments()` in PowerPlant. Each entry in the table lists the type of host object; the message sent; the data sent; the task or function that may be performed immediately after the attachments execute.

Table 15.5 PowerPlant use of attachments

Host	Message	Data	Before
application	`msg_Event`	`EventRecord*`	event dispatch
StDialog Handler	`msg_Event`	`EventRecord*`	event dispatch
commander	the command	command data	`ObeyCommand()`
commander	`msg_Command Status`	`SCommandStatus*`	`FindCommand Status()`
commander	`msg_KeyPress`	`EventRecord*`	`HandleKeyPress()`
commander	`msg_PostAction`	`LAction*`	sending action to super-commander
pane	`msg_Click`	`SMouseDownEvent*`	`ClickSelf()`
pane	`msg_DrawOrPrint`	`Rect*` (frame)	`DrawSelf()`
pane	`msg_DrawOrPrint`	`Rect*` (frame)	`PrintPanelSelf()`
pane	`msg_AdjustCursor`	`EventRecord*`	`AdjustCursorSelf()`

Pay particular attention to the items in the "Before" column. An object may have several attachments. If any attachment returns `false`, these functions *do not execute*. Attachments can control whether event dispatch occurs, panes draw, commanders handle commands, and so forth. This gives each attachment the opportunity to tell your application "I have completely handled this situation, you can ignore it." If all attachments return a value of true, then the host function executes normally, after the attachments completes their work.

TIP: if you call `ExecuteAttachments()` yourself and send the message `msg_AnyMessage`, all attachments will execute.

In the discussion of "Specific PowerPlant Attachments" on page 522 you will see examples of attachments that are designed to respond to various kinds of messages.

Creating your own attachments

Creating an attachment is a fairly straightforward process. Use the PowerPlant attachment classes as examples.

You declare your derived class to inherit from LAttachment. Then you override `ExecuteSelf()`.

The `ExecuteSelf()` function does whatever it is you need to do. Attachments can be designed to respond to a specific message, or a group of messages if you override `Execute()`.

The next sections give you some ideas about when you use attachments, and what you might do with them.

When to use attachments

In general, an attachment is an excellent solution when you have an independent behavior that you wish to implement for a variety of panes or commanders, either in the same project or in different programming projects.

An attachment is also an excellent solution when you want to modify the behavior of a pane or commander dynamically. You can add and remove attachments at will depending upon the application's context.

You can think of an attachment as a kind of inheritance (in a very loose sense) for behaviors. An attachment connects a special function to an object, without the need to create a new class of object. If you think of an object as the sum of itself and its attachments, you can modify the composition of the object dynamically by adding or removing attachments.

Uses for attachments

About the only limitation to attachments is your imagination. The ideas described in this section are but a sampling of what you can do.

Consider the three principal kinds of objects that can host attachments. Applications are commanders and event handlers. Com-

manders handle commands and keystrokes. Panes handle clicks. Some panes are also commanders.

You may design an attachment for event pre-processing. Before any event is ever dispatched, the application's attachments get a crack at it. If ever there was a boundless horizon, this is it. You can do anything you want with the event, and subsequently short circuit event dispatch or allow the event to be handled normally. It's up to you.

A commander's attachments get first crack at all commands. You may design an attachment to handle a specific kind of command. In a traditional approach, your commander's `ObeyCommand()` function handles commands. You might want to create a command-handler attachment that you can connect to any appropriate commander. Then you don't have to duplicate code. If you decide a commander should respond to a particular kind of command, you simply hook up an attachment that does the work.

You could use a command-level attachment to create a demo version of an application. In the demo version, you hook up an attachment that intercepts certain commands—for example the New command—to disable them.

You might design an attachment to handle menu updating. A commander's attachments get first crack at menu update requests as well. PowerPlant provides an attachment for this purpose, LCommandEnabler. Rather than write the code directly into your `FindCommandStatus()` function, you can hook attachments to your commander to handle whatever menu commands you need to take care of.

With respect to panes, attachments have an opportunity to execute before drawing, clicking, and cursor adjustment. You may do some fancy drawing in or around a pane. Perhaps you have some panes that you want to have a fancy border. Create a border attachment and hook it up to those panes. If you want any unique behavior to occur when a pane is clicked, create an attachment to implement the behavior. The possibilities are endless.

Specific PowerPlant Attachments

Looking at some real attachments will help you grasp the power and potential in the attachment design pattern. As we stated before, PowerPlant provides several attachment classes. You can use these whenever appropriate in your own applications. You can also use them as models for your own attachments. In this section we discuss the following attachments:

- LBeepAttachment
- LBorderAttachment
- LEraseAttachment
- LPaintAttachment
- LCommandEnablerAttachment
- LKeyScrollAttachment

All of these classes are declared in UAttachments.h and defined in UAttachments.cp.

The LUndoer class, which derives from LAttachment, is more than a simple attachment. It is the basis for the PowerPlant implementation of undo functionality.

See also *PowerPlant Revealed* for more on action classes and undo.

LBeepAttachment

This is a simple attachment designed to respond to any message you want, typically a click message. When you create the object, you specify the message to which you want the attachment to respond. When attached to any host, this attachment beeps when the appropriate message is received.

LBorderAttachment

This attachment is designed to respond to `msg_DrawOrPrint`. When you create this attachment, you specify a `PenState`, a foreground color, and a background color. You also specify whether the host should draw as well.

This attachment draws a border around the pane's frame with the specified pen.

LPaintAttachment

This attachment is designed to respond to `msg_DrawOrPrint`. When you create this attachment, you specify a `PenState`, a foreground color, and a background color. You also specify whether the host should draw as well.

This attachment paints the host using the specified `PenState` settings and foreground and background colors. The painted rectangle is inset from the pane's frame by the size of the `pnSize` field of the `PenState`. This lets you use an LPaintAttachment in conjunction with an LBorderAttachment to draw a filled rectangle.

Because attachments draw first, you can use this attachment to fill in a background of a pane before drawing occurs.

LEraseAttachment

This attachment is designed to respond to `msg_DrawOrPrint`. It simply erases the pane before drawing.

LCommandEnablerAttachment

This attachment responds to `msg_CommandStatus`. This is a good example of a command-updating attachment. When you create the attachment, you specify the command that should be enabled.

When it executes, this attachment enables the menu item associated with the command. It does not set a mark or do any other item manipulation. It also prevents the host from executing, because it has already enabled the command in question.

You can use this attachment as a model for an attachment that handles other updating tasks. For example, you might create a check mark attachment that puts a check mark in front of a menu item.

LKeyScrollAttachment

This attachment is a good example of keystroke preprocessing. It is also an excellent example of the kind of task for which an attachment is ideally suited.

This attachment handles scrolling a view using keyboard navigation keys: Home, End, PageUp, and PageDown. This kind of functionality is a very nice thing to add to a view. Rather than writing code to implement this functionality in every scrolling view class, why not create an attachment that does the processing for you? Then, if you ever want to add this feature to a view, simply create and connect the attachment.

This particular attachment responds to `msg_KeyPress`. When you create the attachment, you provide a pointer to an LView object. This is the scrolling view. Because this attachment is responding to a keystroke, the attachment must be hosted by a commander.

If you have a view that is also a commander—a class derived from both LView and LCommander—you can attach an LKeyScrollAttachment to it to implement keyboard navigation.

If your view is not a commander, you can attach the LKeyScrollAttachment to a superview that is a commander (such as the window containing the view).

WARNING! If you can delete the scrolling view independently of the commander that has the attachment, you must take care to delete the attachment as well. If you do not, the attachment keeps the pointer to the now-deleted view (a dangling pointer) and you're in for big trouble.

Summary

In this chapter you learned how elegant design can make difficult programming tasks much easier to accomplish.

By the simple expedient of making an object inherit from LPeriodical, you ensure that it receives time on a regular basis. Implementing any time-dependent task becomes trivial. The object receives attention on every pass through the event loop, or at idle time, depending upon whether you install it in the repeater queue or the idler queue. You override `SpendTime()`, and you're done.

PowerPlant's use of attachments reflects an extraordinarily simple, powerful, and unbounded design pattern. This kind of elegance can be found elsewhere in PowerPlant—for example, in the broadcast/listen messaging mechanism. But nowhere else are true power and simplicity so well combined.

You create an attachment, specify the type of message to which it should respond, and override the `ExecuteSelf()` function. You connect the attachment to an appropriate host object. PowerPlant gives your attachment the opportunity to execute at several points in the ordinary flow of events.

Using attachments, you can create independent behaviors that you attach or remove from objects dynamically.

Because this chapter is the end of *the PowerPlant Book*, you'll find a brief recap after the code exercise that sums up where we have been. But first, let's jump into the final code exercise.

Code Exercise

This is it, the goodies you've been waiting for. In this code exercise you get a glimpse at the real power of object-oriented programming with a well-designed application framework.

Best of all, you're going to create two pieces of code that are valuable, real-world additions to your personal collection of reusable code. One is a periodical, and the other is an attachment. Each demonstrates the ease with which you can use both of these marvelous PowerPlant features.

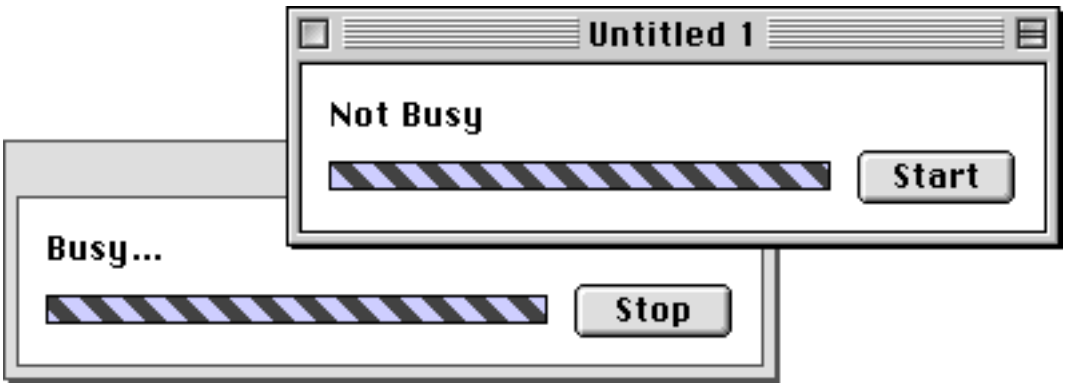
Appropriately, the application you write in this code exercise is titled "Goodies." Let's look at the interface briefly, and then build a periodical and an attachment.

The Interface

In the Goodies application, you create a window that displays a progress bar, as shown in Figure 15.4. This application also has a **Window** menu, just like the menu you built in the code exercise in Chapter 11, "Windows."

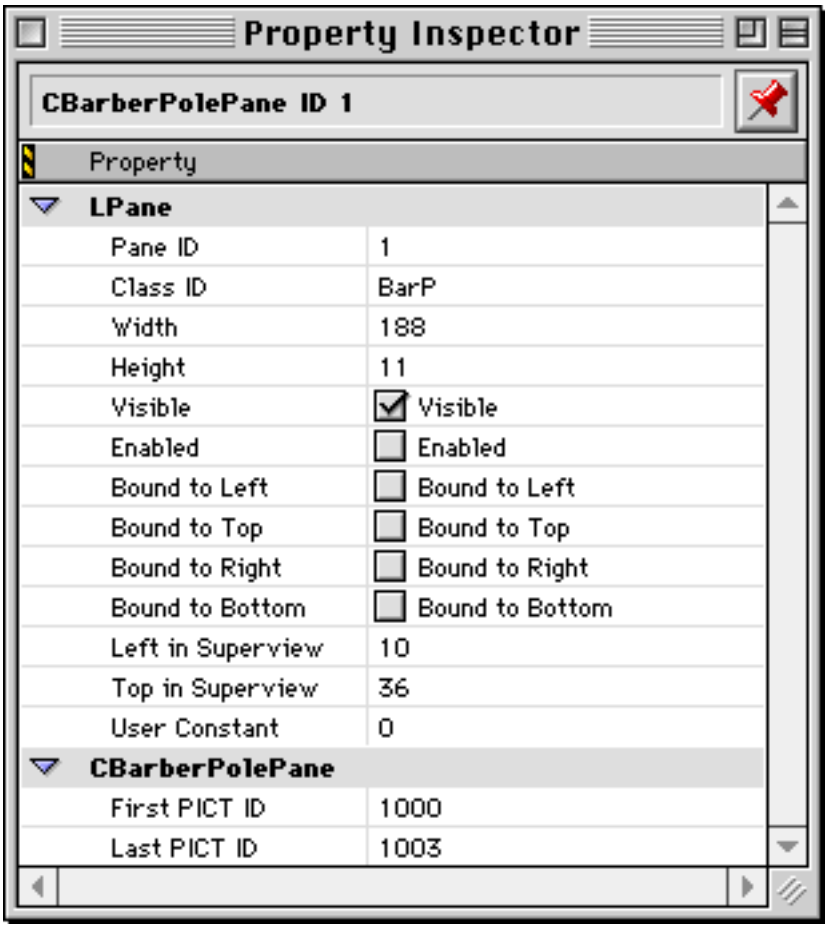
There isn't any real task going on that requires a progress bar, this is just a demonstration. The window has a button to start the bar as if there were something going on. When you click the Start button, the "barber pole" progress bar animates. The text in the button changes to Stop, and the caption changes to Busy.

Figure 15.4 The Goodies window



Open the `Goodies.ppob` project file in Constructor and examine the PPob resource for this window. Pay particular attention to the characteristics of the barber pole pane, as shown in Figure 15.5.

Figure 15.5 Barber pole properties



This pane represents a custom class, with ID BarP. It also has two custom data items, the First PICT ID and Last PICT ID. To animate the barber pole, the pane cycles through a list of PICTs, displaying each picture in turn. In this case there are four PICTs, numbered from 1000 to 1003. The PICT resources have been provided for you in `Goodies.rsrc`.

Feel free to examine the custom pane type—the CTYP resource—as well.You created a custom pane in the code exercise in Chapter 15, "Controls and Messaging."

Implementing Goodies

In this section you implement a periodical task and an attachment. The progress window is a periodical. The **Window** menu is an attachment.

This is the same **Window** menu you created in Chapter 11, so you should be familiar with how it works. The **Window** menu is implemented as a custom class derived from LMenu. The application can't use the PowerPlant default menu-creation mechanism because PowerPlant creates LMenu objects. This application creates a CWindowMenu object and adds it to the menu bar.

In Chapter 11, you added code to the application's `ObeyCommand()` and `FindCommandStatus()` functions to manage the **Window** menu. In this exercise you write similar code, but put it in the *attachment's* `ExecuteSelf()` function! Let's get started.

1. **Examine the CBarberPolePane class**

   `class declaration` `CBarberPolePane.h`

   Look at the code that declares this class. First, notice that this class inherits from both LPane and LPeriodical. It has a class ID BarP. There are several constructors, and a destructor.

   The class overrides the `SpendTime()` function, as is necessary in any descendant of LPeriodical. The class also overrides the `DrawSelf()` function inherited from LPane.

   Finally, notice the data members and the `kThrottleTicks` constant. The object stores the resource IDs of the first, last, and current PICT on display. It stores the time to change pictures in `mNextTime`. It changes the picture every `kThrottleTicks` ticks on the system clock.You'll write the code to do this in the next step.

   Close the file when you are through examining this class.

2. **Animate the barber pole pane.**

   `SpendTime()` `CBarberPolePane.cp`

   The existing code gets the current tick count. After that, you:

   a. Determine if it is time to change pictures.

   Test the current tick count against `mNextTime`.

b. Advance to the next picture.

If it is time to change, increment `mCurrPictID`.

c. Keep the picture in the proper range.

Make sure `mCurrPictID` stays in the range defined by the `mFirstPictID` and `mLastPictID`.

d. Draw the picture.

Call the pane's `Draw()` function.

e. Reset the time to change pictures.

Add `kThrottleTicks` to the current time, and store the result in `mNextTime`.

```
Int32 theCurrTicks = ::TickCount();
if ( theCurrTicks > mNextTime ) {

  // Increment the pict id.
  mCurrPictID++;

  // Rollover if needed.
  if ( mCurrPictID > mLastPictID )
    mCurrPictID = mFirstPictID;

  // Redraw.
  Draw( nil );

  // Get the next time.
  mNextTime = ::TickCount() + kThrottleTicks;
}
```

Take a quick look at the `DrawSelf()` function, just to see what it does. It gets the current picture and draws it. Notice that in `SpendTime()` you call `Draw()`, not `DrawSelf()`. `Draw()` takes care of setup tasks, and calls `DrawSelf()` for you.

When you are through, save your changes and close the file.

3. **Manage the periodical.**

`SetBusyState()` `CProgressWindow.cp`

CProgressWindow is also a custom class. It inherits from both LWindow and LListener. When a CProgressWindow is created, it installs itself as a listener to the Start/Stop button in the window. That button sends a `msg_ProgressControl` message that the window receives in its `ListenToMessage()` function. Feel free to study the code to see how it works.

`ListenToMessage()` calls `SetBusyState()` to do the work, and passes a Boolean value. If the value is `true`, the window is becoming busy and the pane should animate. Otherwise, the window is not busy and the animation should stop. The existing code stores this value in the window's `mBusy` data member. Then:

a. Get the barber pole pane.

The declared constant for this pane ID is `kBarberPolePane`.

b. Turn on the animation if busy.

Install the pane in the idler queue. Use the `StartIdling()` function.

Change the text in the Start button to "Stop." The declared constant for this pane ID is `kProgressControlButton`.

Change the text in the status caption to "Busy…" The declared constant for this pane ID is `kProgressMessageCaption`.

c. Turn off the animation if not busy.

Remove the pane from the idler queue. Use the `StopIdling()` function.

Change the text in the Stop button to "Start." Change the text in the status caption to "Not Busy." See substep b for the names of the constants for these pane IDs.

```
mBusy = inBusy;

// Get the barber pole pane.
CBarberPolePane *theBarberPolePane;
theBarberPolePane = dynamic_cast<CBarberPolePane *>
                        (FindPaneByID( kBarberPolePane ));
```

```
Assert_( theBarberPolePane != nil );

if ( mBusy ) {

  // Start the barber pole idling.
  theBarberPolePane->StartIdling();

  // Set the button title.
  SetDescriptorForPaneID( kProgressControlButton, "\pStop" );

  // Set the message caption.
  SetDescriptorForPaneID( kProgressMessageCaption, "\pBusy…" );

} else {

  // Stop the barber pole idling.
  theBarberPolePane->StopIdling();

  // Set the button title.
  SetDescriptorForPaneID( kProgressControlButton, "\pStart" );

  // Set the message caption.
  SetDescriptorForPaneID( kProgressMessageCaption, "\pIdle" );
}
```

You have now completely implemented the animated barber pole pane. When the user clicks Start, it animates. When the user clicks Stop, it stops. Save your work and close the file.

In the rest of this exercise, you add a **Window** menu attachment to the application. First, we'll examine the **Window** menu code. After that, you have four principal tasks to accomplish. You must install the menu in the menu bar, create an attachment to handle the menu, connect the attachment to the application object, and implement menu functionality in the attachment.

4. **Examine the Window menu.**

   `class declaration` `CWindowMenu.h`

   Examine the member functions in this class. You may recall these functions from Chapter 11, because you wrote some of them. The

`InsertWindow()`, `RemoveWindow()`, `MenuItemToWindow()`, `WindowToMenuItem()`, and `SetCommandKeys()` functions are all identical to the code you wrote in that previous code exercise. Feel free to refer to that chapter for a refresher on the menu operations.

What's new in this file is the declaration of the CWindowMenuAttachment class. It inherits from LAttachment. It has one significant function, `ExecuteSelf()`. It has a single data member—a pointer to the **Window** menu object. You'll write the `ExecuteSelf()` function a little later.

When you're through examining the class declaration, close the file.

5. **Install the Window menu.**

`Initialize()` `CGoodiesApp.cp`

As we mentioned at the start of this section, you cannot rely on the PowerPlant menu-creation mechanism because it creates LMenu objects. The **Window** menu is a CWindowMenu object.

In the application constructor, existing code registers the custom classes. In the `Initialize()` function, you have three tasks to accomplish.

a. Create a CWindowMenu object.

Use the `new` operator. The declared constant for the MENU resource ID is `rMENU_Window`. Store the result in the global variable, `gWindowMenu`.

b. Get the application's LMenuBar object.

Use `LMenuBar::GetCurrentMenuBar()`.

c. Add the new menu to the menu bar.

Use the menu bar's `InstallMenu()` function.

```
// Make the window menu.
gWindowMenu = new CWindowMenu( rMENU_Window );
ThrowIfNil_( gWindowMenu );

// Get the menu bar.
LMenuBar *theMBar = LMenuBar::GetCurrentMenuBar();
ThrowIfNil_( theMBar );
```

```
  // Install the window menu.
  theMBar->InstallMenu( gWindowMenu, 0 );
```

6. **Connect a Window menu attachment to the application.**

   `Initialize()` `CGoodiesApp.cp`

   The code for this step goes right after the code you wrote in the previous step. You have two tasks.

   a. Create a CWindowMenuAttachment.

   Use the `new` operator and create a CWindowMenuAttachment object. You write this constructor in the next step.

   b. Connect the attachment to the application.

   Call the application's `AddAttachment()` function to connect the attachment to the application. Add the new attachment to the end of the attachment list. Specify that the application owns the attachment. This makes the application responsible for deleting the attachment when the application is deleted.

```
theMBar->InstallMenu( gWindowMenu, 0 );

  // Install the window menu attachment.
  CWindowMenuAttachment *theAttachment;
  theAttachment = new CWindowMenuAttachment( gWindowMenu );
  AddAttachment( theAttachment, nil, true );
```

Save your work and close the file.

7. **Define the CWindowMenuAttachment constructor.**

   `CWindowMenuAttachment()` `CWindowMenu.cp`

   To create this attachment, you must do two things.

   a. Call the LAttachment constructor.

   Set this attachment so it responds to any message, and allows the host to execute. This attachment must receive all messages so that it can identify and respond to menu commands and menu updating.

b. Initialize the CWindowMenuAttachment.

Set the `mWindowMenu` data member.

```
CWindowMenuAttachment::CWindowMenuAttachment(
                                    CWindowMenu *inWindowMenu )
  : LAttachment( msg_AnyMessage, true ),
        mWindowMenu( inWindowMenu )
{
}
```

Excellent! You have installed the menu, created the attachment, and connected the attachment to the application. The final task is to implement menu functionality.

In Chapter 11, you did this in the traditional way—by modifying the application's `ObeyCommand()` and `FindCommandStatus()` functions. In the next two steps you implement the same kind of functionality in the `ExecuteSelf()` function of the **Window** menu attachment.

8. **Identify menu update requests.**

`ExecuteSelf()` `CWindowMenu.cp`

Before updating menus, PowerPlant sends a message to the application's attachments. This happens in the `LCommander::ProcessCommandStatus()` function. The message has two parameters. The first is the message itself, `msg_CommandStatus`. The second parameter is a pointer to an `SCommandStatus` structure. That structure holds the data you normally find in the `FindCommandStatus()` parameters.

The attachment receives this message at menu update time. Existing code says that the host should execute, identifies the message received, and defines local variables—including a pointer to an `SCommandStatus` structure.

After that, you:

a. Determine if this is an item you should update.

All commands from the **Window** menu are synthetic, so call the static function `LCommander::IsSyntheticCommand()`. The attachment is not itself a commander, so you must specify the

class. Also make sure the menu ID matches the `mWindowMenu` menu ID. If the item is a synthetic command from the **Window** menu, then you have identified an item that you must update.

b. Get the window object associated with the menu item.

Use CWindowMenu's `MenuItemToWindow()` function.

c. Handle the item here.

If there is a window, set the `mExecuteHost` item to false. You are taking care of the item entirely right here, so the host object (in this case the application) does not need to ask a commander to set this menu item.

d. Set the menu item status.

Set fields in the `SCommandStatus` structure. Enable the item, use a mark, and set the mark to `noMark`. If the window is the top window, set the mark to a check mark. Use `UDesktop::FetchTopRegular()` to identify the top window.

```
SCommandStatus *theStatus = static_cast<SCommandStatus *>
                                        (ioParam);

if (LCommander::IsSyntheticCommand(
    theStatus->command, theMenuID, theMenuItem)
    && theMenuID == mWindowMenu->GetMenuID() ) {

  // Find window corresponding to the menu item.
  LWindow *theWindow =
                  mWindowMenu->MenuItemToWindow( theMenuItem );

  if ( theWindow != nil ) {
    // Handle it's status here.
    mExecuteHost = false;

    // All window items enabled and use a mark.
    *theStatus->enabled = true;
    *theStatus->usesMark = true;
    *theStatus->mark = noMark;

    if ( theWindow == UDesktop::FetchTopRegular() ) {
```

```
      // Check menu item for top regular window.
      *theStatus->mark = checkMark;
    }
  }
}
```

Remember, the attachment is not a commander. There is no inherited `FindCommandStatus()` function for items you don't update. If you don't handle it, the `mExecuteHost` value remains true, and the application asks a commander to take care of updating the item.

Your attachment is only pretending to be a commander, but doing quite a nice job of it. The next thing your attachment must do is obey a command!

9. Handle a Window menu command.

`ExecuteSelf()` `CWindowMenu.cp`

Every item in the **Window** menu has `cmd_UseMenuItem` as the corresponding command number. That means PowerPlant generates a synthetic menu command for each item in the menu.

Before calling the `ObeyCommand()` function, `ProcessCommand()` gives attachments an opportunity to handle a command. At that time it passes the command itself as the message.

The existing code identified the `msg_CommandStatus` message, and you handled that message in the previous step. If the message is not `msg_CommandStatus`, you must:

a. Determine if this is a command you should obey.

Call `LCommander::IsSyntheticCommand()`. Make sure the menu ID matches the `mWindowMenu` menu ID. If both conditions are true (it is a synthetic command from the **Window** menu) then you have identified a command you must handle.

b. Get the window object associated with the menu item.

Use CWindowMenu's `MenuItemToWindow()` function. If you have a window, then you have a command you must handle.

c. Handle the command here.

Set the `mExecuteHost` value to `false`. You are taking care of the command right here, so the host object (in this case the application) does not need to ask a commander to obey this command.

d. Bring the window to the front.

If the window is visible, bring it forward. Use `UDesktop::SelectDeskWindow()`.

```
Int16 theMenuItem;

if (LCommander::IsSyntheticCommand(
    inMessage, theMenuID, theMenuItem )
    && theMenuID == mWindowMenu->GetMenuID() ) {

  // Find the window selected.
  LWindow *theWindow =
                  mWindowMenu->MenuItemToWindow( theMenuItem );

  if ( theWindow != nil ) {

    // Handle the command here.
    mExecuteHost = false;

    // Bring the window to the front.
    if ( theWindow->IsVisible() ) {
      UDesktop::SelectDeskWindow( theWindow );
    }
  }
}
```

Notice once again that you aren't a commander. You can't call an inherited `ObeyCommand()` function for commands you don't handle. If you don't handle it, the `mExecuteHost` value remains true, so the application will ask a commander to obey the command.

Save your work and close the file.

10. Build and run the application.

When the project builds correctly and you run the application, a progress window appears, like Figure 15.4 on page 526. Click the Start button and the barber pole animates. Click Stop and the animation stops. Choose the New item to make more windows. Start them running. This is your periodical task at work. Observe that the animation runs even when the window is in the background.

Examine the items in the **Window** menu. There should be one for each window. The currently active window should have a check mark in front of the item. Choose an item, and the corresponding window should activate.

Each of these utility items—the progress window and the **Window** menu attachment—is a nice bit of reusable code.

The **Window** menu attachment can be dropped into any PowerPlant application. You must make a few changes for it to work. You add the menu to the menu bar when the application launches. You connect the attachment to the application. You modify your window's `FinishCreateSelf()` function to add an item for itself in the window. You modify the window's destructor to remove the corresponding item from the **Window** menu. That's it.

The progress window is highly reusable. Simply create the window whenever you need to display progress. Keep in mind, this particular brand of progress window works at idle time. It will not work to mark progress in a long task that does not return to the event loop. However, it works great when the task whose progress you are indicating is another periodical or regularly returns to the main event loop.

In fact, the CBarberPolePane class could be used more generally as CPictAnimator—displaying any series of PICT's in any appropriate circumstance. You could animate icons, spin arrows, or impement a slide show.

Possibilities for experimentation abound. Make a progress window that draws the percentage complete of a task. The task that requires the progress window should create it, and maintain a connection to the progress window. It can post the percentage complete, and the progress window can draw itself. Play with different ways of representing completeness. You can have a 3-D effect in the bar, an ana-

log clock with a sweep that completes a circle, or a cup that fills with color, just to name a few.

For attachments, create a different kind of menu that provides useful functionality. You might want to try adding a debugging menu that you can use to turn debugging on and off. Perhaps you can implement a font menu as an attachment. You might create a "Demo" attachment that converts an application into a demo version by disabling certain commands. Experiment with PowerPlant's built-in attachments in UAttachment.cp. Try to think of other ways in which you can use attachments.

As always, have a good time exploring. Don't worry about getting lost. You are now ready to head out on your own into the vast spaces of the PowerPlant landscape.

Looking Backward, Looking Forward

It has been a long journey from your first PPEdit application in Chapter 1 to the boundless horizons of attachments. Along the way you have learned a lot about PowerPlant.

You have seen PowerPlant from a high-level that emphasizes the design patterns and principles behind this marvelous application framework. You have seen PowerPlant from the mid-level of classes and functionality, and how objects of various classes work together to implement the design principles. And you have seen PowerPlant from deep inside the code.

You have learned not only what PowerPlant is, but—more importantly—how to use it. After all, isn't that the real goal? You now have all the critical pieces, and you know where they belong in the big picture.

Still, there is more. PowerPlant is not a done deal. PowerPlant is a living, breathing piece of code that continues to grow and evolve. As the Mac OS changes, so too will PowerPlant. Metrowerks wants to keep you at the forefront of technology. The PowerPlant engineers are dedicated to keeping PowerPlant the best Macintosh application framework available anywhere.

Use the *PowerPlant Revealed* HTML reference freely. Browse the PowerPlant code. Read the other PowerPlant documentation available in the CodeWarrior package. And don't forget the appendices to this manual. You'll learn about a wide variety of PowerPlant utilities not mentioned elsewhere in this book.

It is our hope that in these pages you have seen what a truly elegant, powerful, and robust application framework—PowerPlant—can do for you as a programmer. We at Metrowerks want to welcome you to the world of PowerPlant programming.

Congratulations! And may you code in interesting times.

A

PowerPlant Utilities

This appendix covers all the utility classes in PowerPlant.

PowerPlant Utilities Overview

These utilities cover a wide range of services. Some of them are wrapper classes for various Toolbox managers. Others help you save and restore program state in one form or another. Still others provide significant help with common programming challenges like string manipulation, list management, or key filtering.

This appendix is organized by source file. While that might seem an odd way to structure a discussion of various utilities, this approach makes sense for two reasons. First, the PowerPlant designers put related tasks and classes into the same source file. As a result, the source files reflect functional boundaries in programming. Second, identifying the utilities by source file helps you find them more easily. Soon you'll be able to zip directly to the source file you need if you want to look up a function.

In most cases there is both a header file and a source file for each entry in this appendix. However, a single file may contain the declaration or definition for several classes.

The names for some of the more important "utility" classes begin with the letter L. Most utility classes begin with the letter U. In general, the "L" files are more central to PowerPlant and its code. The "U" files are a little more peripheral or limited in their scope. For example, LArray is used throughout PowerPlant for many purposes. UDesktop has specific functions for working with windows.

Finally, this appendix is not a replacement for the *PowerPlant Revealed* HTML reference. The appendix concentrates on how you use these utilities. Consult *PowerPlant Revealed* for complete information on the various data members and member functions in each class.

Classes Discussed Elsewhere

The classes and functions declared and defined in the following files have already been discussed elsewhere in the manual. Please refer to the appropriate section for information about them.

- PPobClasses—See "Register PowerPlant Classes" on page 288.
- UAttachments—See "Attachments" on page 512.
- UDebugging—See "Set Debugging Options" on page 276.
- UDesktop—See "UDesktop" on page 375.
- UEnvironment—See "Check the Environment" on page 287.
- UExceptions—See "Set Debugging Options" on page 276.
- UFloatingDesktop—See "UDesktop" on page 375.
- UMemoryMgr—See "Setup Memory Management" on page 283.
- UModalDialogs—See "StDialogHandler" on page 408 and "Simple Movable Modal Dialogs" on page 410.
- UPrintingMgr—See "Printing Utilities" on page 493.
- URegistrar—See "Register PowerPlant Classes" on page 288.
- UWindows—See "UWindows" on page 374.

More Utility Classes

PowerPlant has many more utility classes designed to help with a wide variety of tasks. Some of these classes have been mentioned in passing. Others have not been mentioned at all. This appendix details the following classes:

LClipboard
LArray
TArray
LArrayIterator
LComparator
UDrawingState
UDrawingUtils
UKeyFilters
UProfiler
UReanimator

LString
LSharable
UResourceManager
UScreenPort
UTextTraits

LClipboard

LClipboard is a descendant of LAttachment. LClipboard is an independent PowerPlant class that can be used in non-PowerPlant projects. However, it does rely on the attachable/attachment design pattern. In typical practice, you'll use LClipboard along with the rest of PowerPlant.

LClipboard supports the global clipboard—the "scrap"—completely. It has all the functions for setting and getting data of arbitrary type and length on the scrap. If you want to implement a local clipboard for use within your application, you must subclass from LClipboard.

Table 15.6 lists the more significant LClipboard functions.

Table 15.6 Some LClipboard functions

Function	Purpose
`GetClipboard()`	return pointer to LClipboard object
`SetData()`	do housekeeping, call `SetDataSelf()`
`GetData()`	do housekeeping, call `GetDataSelf()`
`SetDataSelf()`	put data on the scrap
`GetDataSelf()`	get data from the scrap
`ImportSelf()`	convert scrap to local clipboard
`ExportSelf()`	convert local clipboard to scrap
`ExecuteSelf()`	listen for suspend/resume event, set flags to convert local clipboard

To use LClipboard, you create one, and only one, instance of LClipboard. The `GetClipboard()` function is static, so you can always get a pointer to the clipboard object with `LClipboard::GetClipboard()`.

See also *Inside Macintosh:More Macintosh Toolbox* for information on the Scrap Manager.

Using LClipboard

How you use LClipboard in your code depends greatly on your requirements. For example, do you want to import or export custom data for use with other applications? Do you only want to import or export standard data types (PICT, TEXT, MooV, and 3DMF) for use with other applications? Or do you only want a local clipboard (local scrap) for your own application use?

A global clipboard, or global scrap, allows you to share either standard or custom data between applications.

If you define an application-specific data type to be placed on the global scrap, you need to subclass LClipboard and override `SetDataSelf()` and `GetDataSelf()`. You also need to add some instance variables for storing the custom data. For example, a pointer or handle to your data.

In `SetDataSelf()`, you store the data in your private storage. In `GetDataSelf()`, you retrieve the data from your private storage.

You attach the clipboard object to your application object as shown in Listing 15.1. As an attachment, LClipboard looks for the `msg_Event` message. If a suspend or resume event occurs, LClipboard converts the local clipboard to the scrap or vice versa, as appropriate. It does so by calling `ExportSelf()` or `ImportSelf()` as appropriate.

Listing 15.1 Attaching LClipboard to your application's constructor

```
...
// inside constructor for your application
// setup access to the global clipboard
```

```
AddAttachment( new CMyClipboard );
...
```

To be friendly, if your application's custom clipboard data type can be converted to a standard type, such as TEXT, PICT, MooV, or 3DMF, you need to override `ExportSelf()` and perform the data conversion in that method. If your application can use standard format data copied from other applications, but needs to convert that data to your custom type, you need to override `ImportSelf()` to convert the data to your custom format. Both of these methods are defined empty in LClipboard.

TIP: Generally, `ImportSelf()` would only set a variable saying there is data on the clipboard. You only need to convert the data if the user chooses **Paste**.

An example may be useful here. Say you wrote that killer graphics application. Your application has a custom storage mechanism for your graphic data. **Copy** and **Paste** within your application is not a problem as no data conversion is necessary. However, say a user copies a graph created in your application. The user then switches to another application, maybe a desktop publishing application that only understands the standard clipboard data types, to paste the graph. Your application receives a suspend event. At this time, the `ExportSelf()` method of your LClipboard class is called. You convert your custom data into a format the other application understands in this method. If you do not do this, the other application will not be able to paste the graph.

Similarly, if the user copies a picture from one application and want's to paste it into your graphics application, your application is brought to the front and the `ImportSelf()` method of your LClipboard class is called. If it's more efficient to convert the graphic to your custom format, you can convert the data in this method or wait until the user actually chooses **Paste** and do the conversion then.

If your application only handles the standard format data types, declare an LClipboard object as a member variable of your application subclass like this:

```
LClipboard mClipboard;  // inside declaration of class CMyApp
```

Local Scrap

LClipboard only supports the global clipboard. If you want to maintain a local clipboard (local scrap) for use by your application only, you need to subclass LClipboard and override `GetDataSelf()` and `SetDataSelf()` to use your local scrap instead of the global scrap.

Putting and getting data from the clipboard

To put data on the scrap, you call `SetData()` as shown in Listing 15.2. There are two overloaded versions of this function. One takes a handle to data, the other a pointer and length of data. The actual work is done by `SetDataSelf()`. The default implementation just puts data on the global scrap. In derived classes using a private clipboard, you would override `SetDataSelf()`.

Listing 15.2 Putting text on the clipboard

```
...
// Copy text to the clipboard. Note: clipString is a Pascal string
(LClipboard::GetClipboard())->SetData('TEXT', &clipString[1],
    clipString[0]);
...
```

To retrieve data, you call `GetData()`. You provide a handle. This function fills the block with the data from the scrap. The actual work is done by `GetDataSelf()`. The default implementation just puts data on the global scrap. In derived classes using a private clipboard, you would override `GetDataSelf()`.

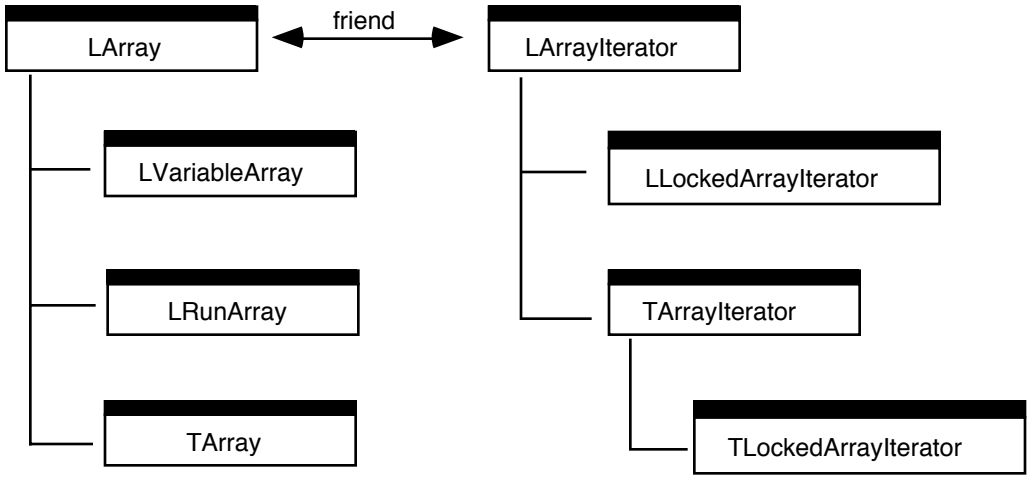
Arrays

Arrays of data are common features of many applications. PowerPlant provides powerful array classes for your use. PowerPlant itself uses these classes in many places.

In PowerPlant the array classes can be organized into three groups: the arrays, the iterators, and the comparators. An array is an indexed series of values. An iterator lets you walk through the items in the array—forwards or backwards, from the beginning, the end, or from an arbitrary location in the array. A comparator is used to sort the contents of the array if you wish to keep the items sorted.

PowerPlant uses its own array classes to manage all kinds of lists of objects. Figure 15.6 shows the class hierarchy for the array classes.

Figure 15.6 Array classes



LArray is the fundamental class for arrays. You can use LArray for an array of any kind of data, with an arbitrary number of items, as long as all items are the same size. LArray is dynamic, which means you can add or remove items from the array freely. LVariableArray allows you to create arrays where each element in the array may contain data of a different size.

LArray and LArrayIterator are each friend to the other. You can iterate over the array or list from an arbitrary starting position, either forward or backward. Iteration works properly even if items are added or removed from the list during iteration. The array iterator does the right thing even if the array disappears completely.

Using LComparator, you can keep the items in an array sorted.

NOTE: In earlier versions of PowerPlant, the LList and LListIterator were used as simple implementations of arrays, as well as used internally within PowerPlant for list management. These classes are now obsolete, so you should update old code to use LArray (or some other appropriate array class). Any new needs for array classes should use LArray, LArrayIterator and their descendants.

The array classes are:

- LArray—the fundamental ordered collection of identically-sized data
- LVariableArray—an array with variable-sized data
- LRunArray—an array where consecutive identical items are stored as a single entry
- TArray—a template-based array class
- LArrayIterator—an iterator to walk an array
- LLockedArrayIterator—an iterator that locks the array
- TArrayIterator—a template-based iterator
- TLockedArrayIterator—a template-based iterator that locks the array
- LComparator—compares elements in an array byte-by-byte
- LLongComparator—compares elements in an array as `long` values

LArray

LArray implements an ordered collection of fixed-size items. Positions in the array are one-based—the first item is at index value 1.

Index 0 is not an item in the array. The index value zero is used to indicate a nonexistent item.

Index values are signed, 32-bit integers. When specifying an item, you pass a pointer to the item data as a parameter. The array stores a copy of the data, or returns a copy of the data to you.

The size of each item in a particular LArray must be the same, but the size can vary between instances of LArray. The actual content of each item can be any type of data—pointers, handles, structures, actual values, and so forth. The only data you should not store in an array (LArray or otherwise) is an object. You should store pointers to objects created via `new()`. One other limitation, specific to LArray, is that each item must use the same amount of storage. You specify the amount of storage per item in the LArray constructor.

Table 15.7 lists some of the functions in LArray.

Table 15.7 Some LArray functions

Function	Purpose
`GetItemSize()`	get data size of each item
`GetCount()`	get number of items in the array
`InsertItemsAt()`	add items at specified index
`RemoveItemsAt()`	remove items at specified index
`Remove()`	remove specified item
`FetchIndexOf()`	get index of specified item
`FetchItemAt()`	get value of item at specified index
`AssignItemsAt()`	set value of items at specified index
`SwapItems()`	interchange two specified index items
`MoveItem()`	move an item from one index to another

You can insert, remove, get a value, assign a value, swap, move items, and get data about the array. There are many more functions in the LArray class. You should consult the source code for details.

NOTE: You cannot store C++ objects in an array. PowerPlant arrays store data using Handles which allow items to move in memory. C++ objects must stay at the same place in memory. Internally, C++ objects store pointers to their various subparts. These pointers are absolute, not relative, so moving the object would produce unpredictable results.

PowerPlant defines the constants `index_First` and `index_Last` so you can easily insert or remove items at the beginning or end of an array. If you attempt to insert an item beyond the current end of the array, PowerPlant inserts the item at the end of the array. If you attempt to remove an item that doesn't exist, PowerPlant does nothing.

The `FetchItemAt()` function has two versions. In one, you specify the size of the data you want returned. You can use this to retrieve partial data from any array element, or control the amount of data returned from an LVariableArray. In typical use you do not need to specify the size of the data.

WARNING! When fetching data from an array, PowerPlant copies the data into a buffer you provide. PowerPlant assumes that the data buffer is large enough to hold the requested data. If it is not large enough, you can expect unexpected results. If you specify a size for the returned data, PowerPlant returns either that amount of data or the actual data in the element, whichever is smaller.

The functions `InsertItemsAt()` and `AssignItemsAt()` also have a default parameter you can use to specify the data size. This allows the same function to work for all arrays, including those with variable size data. If you are working with LVariableArray, specify the size of the data. Otherwise, pass zero or let it default to zero.

You should not specify a size when working with LArray or LRunArray, both of which are arrays with data of one size. If you specify a size of zero, LArray gets the actual size and uses it.

WARNING! When using `InsertItemsAt()`, or `AssignItemsAt()` with LVariableArray, you ***must*** specify the size of the data. If you are working with LArray or LRunArray, do **not** specify the data size.

In typical use, you instantiate an array of items of the desired size. As you add or remove items from the list, you call `InsertItemsAt()` or `RemoveItemsAt()`. These functions take care of notifying any array iterators of changes in the array. When you want to retrieve an item, you call `FetchItemAt()` with the desired index value.

To create a sorted array, you create an LComparator object before creating the array. Then pass a pointer to the LComparator object to the LArray constructor. Or you can use `SetComparator()` to specify an arrays comparator after the fact.

To iterate over an array, you create an iterator object. You pass a pointer to the array to the iterator constructor, so it knows what array to work with.

LVariableArray

LVariableArray is an implementation of LArray that allows you to store data of differing sizes in an array. It overrides several member functions of LArray to implement data storage and retrieval in a situation where array elements vary in size. It also implements a few new functions (all of them internal).

The public interface for LVariableArray is effectively identical to that of LArray. The tasks you need to perform—setting, getting, adding, and removing items in the array—you accomplish by making the same calls you would with an LArray object.

WARNING! When using `InsertItemsAt()`, or `AssignItemsAt()` with LVariableArray, you ***must*** specify the size of the data.

When calling `FetchItemAt()` with LVariableArray, you may wish to specify the size of the data returned if you don't want it all, or you want to ensure that the data returned does not overrun your buffer.

LRunArray

LRunArray is an implementation of LArray in which a contiguous series of identical items (a run of items) is stored once. It overrides several member functions of LArray to implement data storage and retrieval in a situation where a run of array elements is stored in a single element. It also implements a few new functions (all of them internal). You could use LRunArray to save memory in a situation where you could expect to have runs of data.

The public interface for LRunArray is effectively identical to that of LArray. The tasks you need to perform—setting, getting, adding, and removing items in the array—you accomplish by making the same calls you would with an LArray object.

You do not need to keep track of the true index number in the run array. For example, if the first 16 items in an LRunArray are identical, and the 17th is not, the first element in the array holds the data for items 1-16, and the second element in the array holds item 17. However, you deal with the array as if each item occupied a separate element. So you access the 17th item in the array with the index number 17, even though the first 16 items are identical, and therefore stored in a single element.

TArray

TArray is a template-based implementation of LArray. TArray is a subclass of LArray. All functions are one-line inlines that call the corresponding LArray method.

Even though TArray is a template class, it is not that different, in terms of usage, from LArray. The bonus of using TArray of LArray

is that the template implicitly or explicitly performs all the typecasts to and from the `void*` pointers used by LArray. This means that code which uses the template is typesafe.

Furthermore, TArray accepts its arguments as references, unlike LArray which accepts its arguments as pointers.

One caveat to using TArray is that instantiating your TArray will cause inherited virtual functions to be hidden. The compiler will generate a warning about this if you have the "Hidden Virtual Functions" warning turned on. There is no problem doing this. However, to suppress the warning, you can wrap the declaration of the TArray with `#pragma warn_hidevirutal off/reset`.

All of PowerPlant's internal array usage utilizes TArray.

See also The *C Compilers Reference,* and *Assembler Guide* for more information on `#pragma`'s, and the *IDE User Guide* for more information on warning messages.

LArrayIterator

LArrayIterator provides the functionality necessary to walk through an array from an arbitrary starting point, going either forward or backward. Each LArrayIterator object is associated with a single array. An array may have an arbitrary number of iterators, but each iterator has one array.

Rather than use an LArrayIterator, you could walk through the array contents directly. You could call the array's `FetchItemAt()` function and loop through each item. This works fine as long as the number of elements in the array doesn't change.

The iterator is much more robust. The design of LArrayIterator allows for the length of the array to change while iterating, and even for the array to disappear completely. This safety mechanism works as long as you always notify the iterator when the underlying array changes. The implementation of LArray in PowerPlant does this for you.

You can use simple functions in LArrayIterator to traverse the array. LArrayIterator keeps an index value or marker that refers to the current item in the array. Table 15.8 lists the functions of interest.

Table 15.8 LArrayIterator functions for walking a list

Function	Purpose
`Current()`	get item at current marker
`Next()`	get next item in list
`Previous()`	get previous item in list
`ResetTo()`	set the marker to the specified value

If you step past the end of the list, the `Next()` function returns `false`. If you step before the beginning of the list, the `Previous()` functions return `false`.

LArrayIterator has two versions of `Current()`, `Next()`, and `Previous()`. In one version you specify the size of the data you want returned. This is useful if you want only part of the data returned, or with data of varying size. If you are working with LArray or LRunArray, and not using LVariableArray, you typically do not specify a size.

To use an array iterator, you start with the iterator constructor. You specify the array object to which the iterator should be attached, and you set the initial value for the index marker. You can use the constants `from_Start` or `from_End`, or you may specify an exact index number. You can call `ResetTo()` to set the marker at any time.

If you iterate `from_Start`, the marker is set to non-existent item zero. Call `Next()` to get the first item in the array. Conversely, if you iterate `from_End`, call `Previous()` to get the last item. Listing 15.3 shows you how to iterate from the start to the end of an array.

Listing 15.3 Iterating from the start of an array

```
{
  // Iterating from beginning to end of myArray
  LArrayIterator iterator(myArray,LArrayIterator::from_Start);
  while (iterator.Next(&theItem))
  {
    // do something with theItem
  }
}
```

Remember that the iterator is a separate class from the array. You can have multiple iterators for the same array. For example, Listing 15.4 shows how to remove duplicate entries from an array using two iterators simultaneously on the same array.

Listing 15.4 Multiple iterators for a single array

```
{
  LArrayIterator outer(myArray, LArrayIterator::from_Start);
  while (outer.Next(&testItem)) {
    LArrayIterator searcher(myArray,
                            myArray->FetchIndexOf(&testItem));
    while (searcher.Next(&searchItem)) {
      if (testItem == searchItem) {
        myArray->Remove(searchItem);
      }
    }
  }
}
```

The outer iterator starts from the beginning of the array. The searcher iterator starts after the position of the current item in the outer iterator and removes any item that matches that item. Each iterator moves properly to the next item in the array, even when an item is removed.

The PowerPlant source code is replete with examples of arrays and array iterators. Browse the code to see how it's done. Specifically,

the "LArray Demo," located on the CodeWarrior Reference CD, demonstrates the basics of how to use the array classes.

LLockedArrayIterator

LLockedArrayIterator is a subclass of LArrayIterator that locks the array before traversing it. This is useful when accessing pointers to items in arrays that do not change while iterating, or any time you might otherwise need to lock an array.

Usage of LLockedArrayIterator is no different than using an LArrayIterator. Since it is the constructor that locks the array and the destructor that unlocks it, gaining the benefits of LLockedArrayIterator is seamless.

NOTE: Since the locking and unlocking occur in the constructor and destructor, you need to ensure the LLockedArrayIterator is destroyed before its associated array is destroyed. If you create the LLockedArrayIterator on the stack, you can simply use braces to limit the scope and life of the LLockedArrayIterator object.

TArrayIterator

TArrayIterator is a template-based implementation of LArrayIterator. TArrayIterator is a subclass of LArrayIterator. The entire class is declared as inlines in `TArrayIterator.h`.

TArrayIterator is to LArrayIterator as TArray is to LArray. The relationships are analogous. It is typesafe, and uses references instead of pointers.

TArrayIterator is used throughout PowerPlant itself. Reading the source code will show how to utilize the class. Specifically, the "LArray Demo" on the CodeWarrior Reference CD demonstrates how to use this class.

TLockedArrayIterator

TLockedArrayIterator is a subclass of TArrayIterator. It functions the same as TArrayIterator as well as mirroring the same locking functionality of LLockedArrayIterator.

LComparator

LComparator is a simple class. LComparator objects know how to compare two objects or structures. LComparator has four member functions.

Table 15.9 LComparator functions

Function	Purpose
`Compare()`	compare two items
`IsEqualTo()`	returns true if two items are the same
`CompareToKey()`	compare data with a key
`IsEqualToKey()`	returns true if data matches a key

The `Compare()` function should return a value less than zero if item 1 is less than item 2, zero if they are the same, and greater than zero if item 1 is greater than item 2. `CompareToKey()` should do the same against the key value. `CompareToKey()` is not implemented. If you wish to compare against a key, you must override LComparator.

LComparator does a byte-level comparison. It uses the `BlockCompare()` function defined in `UMemoryManager.cp` to do the work.

To create a sorted array, you first create an LComparator object. You then pass a comparator pointer to the array constructor. PowerPlant takes care of the rest. It keeps the array sorted as you insert new items. Removing items does not affect sorting.

The "LArray Demo" on the CodeWarrior Reference CD demonstrates how to use LComparator.

LLongComparator

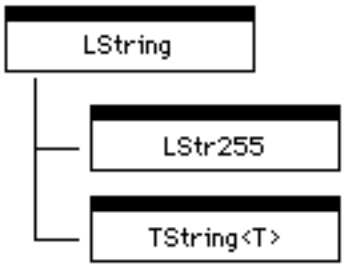
LLongComparator is a subclass of LComparator. It assumes that the items being compared are `long` values. LLongComparator overrides `Compare()` and `IsEqualTo()`. However, you use LLongComparator exactly as you would LComparator.

LLongComparator is declared in `LComparator.h` and defined in `LComparator.cp`.

LString

The LString class implements string functionality for Pascal-style strings. It serves as a base class for two PowerPlant string classes, LStr255 and TString. Figure 15.7 illustrates the class hierarchy.

Figure 15.7 LString class hierarchy



TString is a template class. You can use it as a basis for a string of any type (an array of unsigned chars). LStr255 is a Pascal-style string with a maximum of 255 characters.

Use *PowerPlant Revealed* to get details on these string classes and their functions. LString is a powerful class with many features.

LString has functions or operators to:

- Convert a string to a number—long, floating point, or a four-character code.
- Fill in the contents of a string from a variety of sources, including:
  - a substring of another string

  - a character
  - a four-character code (e.g. an `OSType`)
  - a pointer to data
  - an STR or STR# resource
  - a long number
  - a floating point number
- Find a substring within a string, including functions to start from either end, or determine if the string begins or ends with a specified substring.
- Insert, remove, or replace parts of the string.
- Compare strings.
- Compare strings with overloaded operators `==`, `!=`, `>`, `<`, `>=`, and `<=`
- Copy a string.
- Copy a string with overloaded `operator=`.
- Append strings.
- Append strings using overloaded operators `+` and `+=`.

If you are going to do significant work with strings, examine the PowerPlant LString class and its descendants. These classes are independent of the rest of PowerPlant and can be used without any other PowerPlant classes.

NOTE: LString replaces the functions found in the now-obsolete String_Utils file. You can use String_Utils for some simple work. There are functions to copy or concatenate Pascal strings. There are also functions to convert between a Pascal string and an OSType (four-character code).

LSharable

LSharable is a mix-in class to allow objects to delete themselves when no longer used. LSharable uses a reference counter to keep track of the number of objects currently using the shared data. Once

the counter reaches zero, the class deletes itself, thus freeing up any memory that was used by the object.

An example of where you might use LSharable is a database application where you may have several different views of the same shared data object. Once the last view no longer needs to see the data, the data object deletes itself.

UScreenPort

PowerPlant uses UScreenPort on certain occasions to manage the desktop. This class creates a GrafPort that is the same size as the gray region—the area of all monitors, excluding the menu bar. If the port hasn't been created when you try to access it, the class creates the port automatically.

UDrawingState

The UDrawingState files declare and define several PowerPlant utility classes. One of them is UQDGlobals.

The UQDGlobals class has four functions, listed in Table 15.10.

Table 15.10 UQDGlobals functions

Function	Purpose
`InitializeToolbox()`	initialize the basic Toolbox managers
`GetCurrentPort()`	return the current GrafPort
`GetQDGlobals()`	return pointer to QDGlobals
`SetQDGlobals()`	set QDGlobals

The remaining classes in UDrawingState are designed to preserve and restore drawing state information. To use any of these classes, you simply define a local, stack-based object of the class. The constructor preserves the information. When the function goes out of scope, the class destructor is automatically called. The destructor restores the original state.

Using these classes makes saving your drawing state virtually automatic and foolproof. Even if the function terminates abnormally—for example, because of an exception—the correct destructor is called and state is restored. Table 15.11 lists each of these classes, and the data they preserve and restore.

Table 15.11 Stack-based drawing state classes

Class	Preserves/Restores
`StColorState`	foreground and background color
`StColorPenState`	pen location, size, mode, pattern, and color state
`StTextState`	font number, text size, style, and mode
`StClipRgnState`	clipping region
`StPortOriginState`	port origin
`StColorPortState`	GrafPort, port origin, pen state, color state, text state, and clipping state
`StHidePen`	hides and shows the pen
`StEmptyVisRgn`	current visRgn; sets visRgn to empty, preventing drawing

Most of these classes also have a `Normalize()` function to set the values for that class's data to default values. For example, the `StColorState::Normalize()` function sets the foreground color to black and the background color to white.

The StClipRgnState class has additional constructors for setting a new clipping region, and for other clip region manipulations. Browse the PowerPlant source code to see these classes in action. Use *PowerPlant Revealed* to learn more about them. None of these classes is dependent upon PowerPlant. You can use these classes in any C++ code you write.

See also "Initialize the Toolbox" on page 283.

UDrawingUtils

The UDrawingUtils files declare and define four classes, each related in some way to drawing. We'll discuss each class in turn. They are:

- UDrawingUtils
- StDeviceLoop
- UMarchingAnts
- UTextDrawing

UDrawingUtils

UDrawingUtils declares three static functions. Because every function is static, you never declare an object of this class. The class is simply a device for grouping these functions. Table 15.12 lists them.

Table 15.12 UDrawingUtils functions

Function	Purpose
`IsColorGrafPort()`	return whether specified port is a color port
`IsActiveScreenDevice()`	return whether specified GDevice is active
`SetHiliteModeOn()`	turn on QuickDraw highlight mode

You may find these functions useful when drawing. None of them is dependent upon any other part of PowerPlant.

StDeviceLoop

The StDeviceLoop class is designed to assist you when drawing items that are color-depth-sensitive on multiple monitors.

Like other stack-based classes, you declare a local object of the class. Typically you would do this in a pane's `DrawSelf()` routine. How-

ever, the implementation is not pane-dependent. You can use this class independently of PowerPlant.

The constructor takes a `Rect` in the local coordinates of the current port. Typically, this would be the pane's frame.

The `NextDepth()` function passes back the depth of the next device and returns true. After reaching the last device, the depth is zero and the function returns false. You would normally call `NextDepth()` as the condition in a "while" loop.

When `NextDepth()` returns with a valid depth, it has already set the clipping region to the intersection of the specified `Rect`, the current device, and the original clipping region. Therefore, you can just draw the pane and rely on the clipping region to properly restrict the drawing. If you need access to the current device, you can call the `GetCurrentDevice()` member function.

The destructor restores the clipping region to what it was when the constructor was called.

Listing 15.5 shows some example code using StDeviceLoop.

Listing 15.5 A typical use of StDeviceLoop

```
Rect  frame;
if (CalcLocalFrameRect(frame)) {
  StDeviceLoop theLoop(frame);
  Int16 depth;
  while (theLoop.NextDepth(depth)) {
    switch (depth) {

      case 1: // Black & white
        break;

      case 4: // 16 colors
        break;

      case 8: // 256 colors
        break;
```

```
      case 16: // Thousands of colors
        break;

      case 32: // Millions of colors
        break;
    }
  }
}
```

You supply the appropriate drawing code for the different color depths.

UMarchingAnts

The UMarchingAnts class provides some support for a standard Macintosh animated selection marquee. All of the members of this class are static, so you do not have to instantiate an object of this class.

The `UMarchingAnts::BeginMarch()` function sets up a pen pattern for the marquee. `UMarchingAnts::EndMarch()` cleans up. In between you are responsible for managing the marquee—erasing, resizing, and drawing the marquee as the mouse moves.

UTextDrawing

The UTextDrawing class has a single static member function, `DrawWithJustification()`. PowerPlant uses this function to draw text in LCaption and LTextButton objects.

`DrawWithJustification()` provides the same functionality as the Toolbox `TextBox()` routine, with one exception. The PowerPlant function does not erase the box before drawing. This enhances performance.

However, if you change the text in a caption or text button dynamically, you must erase the text yourself before drawing the new text.

UKeyFilters

The UKeyFilters class defines three different key filters. Every function in this class is static, so you never have to declare a UKeyFilter object. You can use the functions at any time. Even better, UKeyFilters is another independent PowerPlant module that you can use in any project.

In PowerPlant, a key filter examines a keydown event and returns a value to you that tells you something about the key. You can then act based upon the value you receive from the filter. The filter is an automatic way of testing a key to see if it fits certain criteria.

The return value from each filter is an enumerated data type, `EKeyStatus`. The possible `EKeyStatus` values are summarized in Table 15.13.

Table 15.13 EKeyStatus values

Constant	Meaning
`keyStatus_Input`	input character is acceptable
`keyStatus_TEDelete`	TextEdit delete key
`keyStatus_TECursor`	TextEdit cursor movement key
`keyStatus_ExtraEdit`	edit key not supported by TE
`keyStatus_Ignore`	do nothing with the keystroke
`keyStatus_Reject`	invalid keystroke
`keyStatus_PassUp`	pass keystroke to next handler

Each call to a filter returns one of these values depending upon the character in the event record. The filter doesn't tell you precisely what key is in the event. It does tell you whether the key passes the filter, something about the nature of the key, or what to do with the key.

The UKeyFilters class includes three key filters.

Table 15.14 UKeyFilters filter functions

Function	Allows
`IntegerField()`	numbers 0-9
`AlphaNumericField()`	numbers and letters A-Z and a-z
`PrintingCharField()`	any printing character

A "printing character" is a character with an ASCII value from 32 to 126. Each of these functions is static.

You can use Constructor to assign one of these three key filters to an LEditField object. You can set a key filter at runtime using `LEditField::SetKeyFilter()`. You provide a function pointer to the static filter function.

This mechanism allows you to create and use your own filter functions in a class derived from UKeyFilters. When you create an LEditField object, you call `SetKeyFilter()` to attach the desired filter to the object.

The three PowerPlant filter functions rely on several lower-level routines to process characters. You can use these functions directly for "quick and dirty" character testing, or as utilities in your own key filter. Table 15.15 summarizes the available functions.

Table 15.15 UKeyFilters character testing functions

Function	Identifies
`IsTEDeleteKey()`	delete key
`IsTECursorKey()`	up, down, left, right arrow keys
`IsExtraEditKey()`	home, end, page up, page down, forward delete, clear keys
`IsNavigationKey()`	home, end, page up, page down, arrow keys
`IsActionKey()`	enter, tab, return, escape keys

Function	Identifies
`IsNonprintingChar()`	ASCII value 0 to 31 or forward delete key
`IsPrintingChar()`	ASCII values 32 to 126
`IsNumberChar()`	0-9 keys
`IsLetterChar()`	A-Z and a-z keys
`IsCmdPeriod()`	the command-period combination

The `IsCmdPeriod()` function supports international keyboards.

You can find character-related constants declared in the `PP_KeyCodes.h` file.

UProfiler

The UProfiler files declare and define one class, StProfileSection. This is a simple, stack-based utility class to facilitate profiling a section of code using the CodeWarrior Profiler. UProfiler is an independent PowerPlant module that you can use in any project.

To use StProfileSection, you must have the project preferences set up for profiling, and the correct Profiler library included in the project. If you are set up for profiling, StProfileSection makes profiling extremely simple.

StProfileSection has two functions—a constructor and a destructor. Define a local StProfileSection variable before making the function call you want to profile. You provide a file name for the profiler's dump file, the number of functions to be profiled, and the expected stack depth (the nesting depth of function calls). The constructor initializes and activates the Profiler.

You can profile an entire application by creating the StProfileSection object in the `main()` function before telling the application to run.

When your StProfileSection object goes out of scope, the destructor automatically dumps results to the Profiler dump file. You would

then use the Profiler to view the results. Please read the Profiler Manual for details.

See also the *CodeWarrior Profiler Manual* for details on setting up a project for Profiling and the *PowerPlant Advanced Topics* chapter on profiling PowerPlant code.

UReanimator

PowerPlant uses this class internally to build pane objects from a PPob resource. In typical PowerPlant programming, you don't have to deal with this class at all, with one exception.

The `UReanimator::LinkListenerToControls()` function connects a listener to the controls in a RidL resource.

See also "RidL Resource" on page 116 and "Linking broadcasters to listeners" on page 228.

UResourceManager

The UResourceManager files declare and define three stack-based classes for managing resources. UResourceManager is an independent PowerPlant module that you can use in any project. To use it, the only other PowerPlant files you need are the UMemoryMgr files.

We'll discuss each class in turn. They are:

- StNewResource
- StDeleteResource
- StResLoad

You should also review the discussion of the StResource class described in "Stack-based memory classes" on page 285.

StNewResource

You use StNewResource to create a new resource, or modify an existing resource.

Like other stack-based classes, you instantiate a local object. The constructor gets the handle to the existing resource, if it exists. Otherwise it allocates a new handle for you.

After you create the object, you modify the pre-existing resource or write new data into the handle provided for you by StNewResource.

When the local object goes out of scope, the destructor writes the resource to the resource fork, and releases the handle.

StDeleteResource

You use StDeleteResource to remove a resource from the resource fork of a file. The constructor gets the resource handle for the specified resource. The destructor removes the resource from the file and releases the resource handle.

StResLoad

You use StResLoad to preserve, change, and restore the ResLoad parameter in low memory. Typically you would do this to turn ResLoad off temporarily, then restore ResLoad when your operation is complete.

UTextTraits

The UTextTraits class provides support for managing the appearance of text. The UTextTraits class is a fairly independent PowerPlant module. It requires UEnvironment and UMemoryMgr.

All of the functions in UTextTraits are static. You never instantiate a UTextTraits object. You can use the functions at any time.

PowerPlant declares a `TextTraitsRecord` to store the following text characteristics:

- font name
- font number
- text size
- text style

- text justification
- text drawing mode
- text color

The font number is determined from the font name at runtime.

The same information may be stored in a Txtr resource. You can use UTextTraits to work with either a Txtr resource, or with a `TextTraitsRecord` in memory. Table 15.16 lists all the UTextTraits functions.

Table 15.16 UTextTraits functions

Function	Purpose
`LoadSystemTraits()`	set text traits to system default values
`LoadTextTraits()`	get text traits
`SetPortTextTraits()`	set port text characteristics
`SetTETextTraits()`	set TextEdit record characteristics

Except for `LoadSystemTraits()`, there are two overloaded versions of each of these functions: one for working with a `TextTraitsRecord`; the other for working with a Txtr resource.

When you initialize a `TextTraitsRecord` in memory, set all the values directly—including the font name. However, set the font number to `UTextTraits::fontNumber_Unknown`. This is the value -1. Then call `LoadTextTraits()`. This function looks up the font number for the named font.

If you are working with a Txtr resource, call `LoadTextTraits()`. It reads the resource and (assuming you saved the resource with the value -1 as font number) gets the font number for the named font. It puts the results in a handle-based `TextTraitsRecord`.

See *PowerPlant Revealed* and source code for details about these and other UTextTraits functions.

B

Resource Notes

This appendix covers the various resources used in PowerPlant. There are three principal areas to cover:

- PowerPlant-Specific Resources
- Standard Resources
- ToolServer and Rez

PowerPlant-Specific Resources

PowerPlant has several specific resource formats, including PPob, RidL, Mcmd, and Txtr. We have discussed each of these resources at various places in this manual.

For general information on all of these resources, and how to install resource templates for 3rd party resource editors, see "Installing Resource Templates" on page 61.

For more information on the PPob resource, especially the text format of PPobs, see the `PowerPlant.r` file. We discussed various aspects of the PPob resource throughout this manual.

For more on the RidL resource, see "Linking broadcasters to listeners" on page 228.

For more on the Mcmd resource, see "Menu-Related Resources" on page 317.

For more on the Txtr resource, see "UTextTraits" on page 569.

Standard Resources

PowerPlant provides a variety of resources in several files. Most of these resources are standard resources in the Mac OS. A few are custom resource types.

In this section we look at the contents of those files. All of these files are in the PowerPlant Resources folder.

The files discussed are:

- PP Copy & Customize.ppob
- PP Copy & Customize.rsrc
- PP Action Strings.rsrc
- PP DebugAlerts.rsrc
- PP Document Alerts.rsrc
- PP AppleEvents.rsrc
- ColorAlertIcons.rsrc

PP Copy & Customize.ppob

This file is copied, renamed, and used as part of the PowerPlant stationery projects. In typical use, you open and rename this file, and use it as the basis for further development. You can add resources to this file, modify the existing resources, or create additional resource files for your project.

MBAR—Menu Bar

Standard resource for specifying the MENUs in a Menu Bar.

The default constructor for LApplication uses this resource to create the initial menu bar for a program. You should change this resource to contain the ID numbers of the MENUs contained in your program's menu bar.

MENU—Menu

Standard resource for Toolbox Menus.

- 128 Apple
- 129 File
- 130 Edit

Mcmd—Menu Command

Custom resource type for specifying the command numbers associated with menu items. An Mcmd contains a list of 32-bit numbers corresponding to the items in the MENU with the same ID number.

- 128 Apple
- 129 File
- 130 Edit

STR#—String List

- 200 Standards

Common strings used by PowerPlant.

– 1"MyProgram"

Change this to the name of your program. PowerPlant uses this string when it displays the program's name in a dialog box.

– 2"Save File As:"

This is the prompt string displayed in the standard file dialog box for saving a file.

Txtr—Text Traits

PowerPlant text traits resources.

- 128 System Font
- 129 App Font

PP Copy & Customize.rsrc

This file is copied, renamed, and used as part of the PowerPlant stationery projects. In typical use, you open and rename this file, and use it as the basis for further development. You can add resources to

this file, modify the existing resources, or create additional resource files for your project.

aete—Apple Event Terminology Extension

This is a standard resource that defines the natural language syntax of Apple events supported by a program. This information is used by script editors.

You need to have a separate aete for each human language you wish to support, such as English. The ID number of the resource specifies the language. Check the Apple events documentation from Apple for a list of ID numbers and the languages to which they correspond.

We provide an aete for the English language (ID = 0) that specifies the terminology for all Apple events supported by PowerPlant. To properly support script editors, you must change this resource to reflect the Apple events actually supported by your program.

ALRT—Alert Box

`LApplication::ShowAboutBox()` displays this Alert when the user chooses the About item from the **Apple** menu. If you wish to use a simple Alert for your About Box, change this resource (and its associated DITL) as appropriate.

You do not need this resource if you override `LApplication::ShowAboutBox()` to display your program's About Box and use some other alert, dialog, or window.

There is an additional alert for a low-memory warning.

DITL—Dialog Item List

Standard resource for items in the Alert described above.

PP Action Strings.rsrc

Resource file with STR# resources for "undo" and "redo." These strings are used by LAction and LUndoer to change the text of the **Undo** menu item.

This file is included in the PowerPlant stationery. If you do not use stationery when you make a new project and you use LEditField, add this file to your project.

PP DebugAlerts.rsrc

Contains ALRT and associated DITL resources used during debugging.

- 251 ThrowAt

  Alert displayed when an exception is thrown, `Debug_Throw` is defined and `gDebugThrow == debugAction_Alert`.

- 252 SignalAt

  Alert displayed when a signal is raised, `Debug_Signal` is defined and `gDebugSignal == debugAction_Alert`.

For more information, see "Set Debugging Options" on page 276.

This file is included in the PowerPlant stationery. If you do not use stationery when you make a new project, add this file to your project.

PP Document Alerts.rsrc

Contains ALRT and associated DITL resources for confirmation dialogs used in the LDocument classes. Be sure to add this file to your project if you use these classes. The PP_Resources.h file contains the defines for these resources.

- 201 Save Before Closing
- 202 Save Before Quitting
- 203 Confirm Revert

PP AppleEvents.rsrc

Contains the aedt resource that are required for full Apple Event support. Note that this file does not contain the aete resource.

aedt—Apple Event Dispatch Table

- 128 Required Suite
- 129 Core Suite
- 130 Misc Standards

This is a custom resource type (also used by MacApp) for associating a 32-bit number with a particular Apple event. The Toolbox identifies Apple events with a pair of 32-bit numbers (Class ID, Event ID). It is inconvenient to use two numbers to identify Apple events in code, so we use an aedt to map from the two numbers to just one.

It's not necessary, but we use a separate aedt for each Apple event suite. You should define new aedt resources if your program supports additional Apple events.

`UAppleEventsMgr::InstallAEHandlers()` installs an Apple event handler for every entry in every aedt resource included in the program.

ColorAlertIcons.rsrc

You can add this resource file to any program and it will colorize the standard alert icons.

cicn—Color Icon

- 0 Icon used by StopAlert
- 1 Icon used by CautionAlert
- 2 Icon used by NoteAlert

These icons are color versions of the standard System icons displayed by the Alert calls. If you include these cicn resources in your program, the System will use them when displaying Alerts on color screens. If you don't include them, the System uses the standard

black and white icons. We think the color icons are more attractive. However, each cicn is about 1K in size, so there is some space penalty for using them.

ToolServer and Rez

This section describes how to use Rez and DeRez to work with PowerPlant resource files. To install Rez and Toolserver, see the CodeWarrior User's Guide and the Rez Documentation folder on the CodeWarrior CD. CodeWarrior also has a plug-in Rez compiler.

Using ToolServer

You run ToolServer by choosing **Start ToolServer** from the **Tools** menu in the CodeWarrior IDE. This will launch the ToolServer program and display a ToolServer worksheet window within CodeWarrior.

ToolServer has a command line interface. You type commands into the ToolServer Worksheet. To execute a command (or several commands) select the lines containing the command(s) and press the Enter key or Command-Return. Just pressing Return creates a new line (as with a normal text editing window).

Alternatively, you can create a text file that contains ToolServer commands. Then you can execute all those commands by opening that text file in CodeWarrior and choosing **Execute as a Script** from the **ToolServer** menu.

Rez

Rez is a tool that compiles text representations of resource data into actual resources. For examples of defining resources as text, see the various .r files in the "PowerPlant Cookbook" folder.

You will normally use Rez only for resources that have convenient text representations. This includes all resources whose data is primarily strings or numbers. Graphical resources—icons and pictures for example—are best edited with visual tools such as ResEdit and

Resorcerer. However, PowerPlant's PPob resources fit quite nicely into the Rez text format.

A typical Rez command line looks like this:

```
Rez -o "HD:Projects:MyProgram.µ.rsrc" -a
"HD:Projects:MyProgram.r"
```

In this command, `-o "HD:Projects:MyProgram.µ.rsrc"` specifies the output file for the Rez operation. The "`-a`" option means to append or merge the resources into an existing file. Without the `-a` option, Rez will overwrite the output file. Finally, `"HD:Projects:MyProgram.r"` is the input file containing the text defining the resources.

By convention, files containing Rez resource definitions have a ".r" extension.

The above command line does not specify the files that define the format of the resource types, such as Types.r or PowerPlant.r. It is assumed that the necessary definition files are included at the top of the MyProgram.r file.

For example,

```
#define SystemSevenOrLater
#include $$Shell("RIncludes") "PowerPlant.r"
#include $$Shell("RIncludes") "Types.r"
#include $$Shell("RIncludes") "SysTypes.r"
```

inside MyProgram.r will include the proper files. ToolServer defines the Shell variable `RIncludes` to be the full path name to the folder containing the Rez interface files. See the file "StartupTS" in the ToolServer folder to see how it defines its Shell variables. Defining the symbol "SystemSevenOrLater" lets you use certain System 7 specific resource formats.

The file "Rez Script" contains a simple script that uses the GetFileName tools to prompt the user for the input and output files for Rez (as opposed to having to specify the full path names).

For a more complex example, use the "Build Resource Files" script file in the "PowerPlant Cookbook" folder.

DeRez

DeRez is the inverse of Rez. DeRez decompiles resources into text representations of the resource data.

A typical DeRez command line looks like this:

```
DeRez "HD:Projects:MyProgram.μ.rsrc" -only PPob "PowerPlant.r" >
"HD:Projects:MyDeRez.r"
```

This command decompiles resources of type `PPob` in the "HD:Projects:MyProgram.μ.rsrc" resource file. It uses the resource definitions in "PowerPlant.r" to interpret the resource data. It places the output text data in the file "HD:Projects:MyDeRez.r". If the output file does not exist, it is created. If the output file already exists, it is overwritten.

The `-only PPob` option specifies that only PPob resources in the input file are decompiled by DeRez. You can specify as many `-only` options and resource definition files as you want. For example,

```
DeRez "ResFile" -only vers -only MENU "Types.r" "SysTypes.r" >
"MyDerezFile.r"
```

If you don't include any `-only` parameters, then all resource types in the input file are decompiled by DeRez.

Index

A

Activate()
  LPane 148
  LWindow 371
  UDesktop 376
  UDesktop in dialogs 411
ActivateSelf()
  LPane 145, 148
  LWindow 370
active state 132
AddAttachment()
  LAttachable 516
AddListener()
  LBroadcaster 228, 229, 232
AddMenu()
  LMenuBar 332
AddSubCommander()
  LCommander 309
AdjustCursorSelf()
  LPane 148
AdjustScrollBars()
  LView 190, 192
adorner as MacApp class 514
aedt resource in PowerPlant 292
alignment in placeholder 478
AllowSubRemoval()
  LCommander 309
  LDialogBox 408
  LSingleDoc 443
  LWindow 374
AllowTargetSwitch()
  LCommander 311
Apple events
  in LDocApplication 438
  in PowerPlant 291, 292
application
  and main() function 275
  as commander 273
  as design pattern 80
  as event dispatcher 272
  as scriptable object 272
  as top commander 291, 309
  checking environment 287
  class hierarchy 104, 272
  deriving 274
  event handling 290
  functions 273
  implementing 450
  initalize heap 282
  initialize toolbox 283
  initializing 275–290
  memory management 283–287
  overriding functions 274
  quitting 274
  registering classes 288
  running 290
  state 273
application framework
  advantages 77
  and interface 76
  defined 76
  design patterns in 78–88
  flow of control in 77
  utilities in 88
arrays 547–558
  iterating 553
  multiple iterators 555
AskForOneNumber()
  UModalDialogs 410
AskForOneString()
  UModalDialogs 410
AskPageSetup()
  UPrintingMgr 493
AskPrintJob()
  UPrintingMgr 493
AskSaveAs()
  LDocument 441, 453
Assert_ macro 279
  and side effects 279
AssignItemsAt()
  LArray 549, 550
  LVariableArray 551, 552
attachment 512–524
  creating 520
  defined 512
  messages and data 519
  strategy 514
  uses for 520
  when PowerPlant calls 518
  when to use 520
AttemptClose()

LDialogBox 408
LDocument 441
LWindow 372, 373
AttemptQuit()
LCommander 309
AttemptQuitSelf()
LCommander 309, 310
LDocument 441

B

BeginDrawing()
LGWorld 369
BeginMarch()
UMarchingAnts 564
behavior as MacApp class 514
BeTarget()
LCommander 311, 314
binding in a frame 147
BlockCompare() 286
used in LComparator 557
BlocksAreEqual() 286
broadcaster
defined 86
in design pattern 86
linking to listener 228, 232
broadcasting a message 230
BroadcastMessage()
LBroadcaster 228, 230
LControl 231
listen in dialogs 406
BroadcastValueMessage()
LControl 230, 231
listen in dialogs 406
button See control
buttons in PowerPlant 216

C

CalcLocalFrameRect()
LPane 146, 193
CalcPortFrameRect()
LPane 146, 193
CalcRevealedRect()
LView 190
calling Toolbox routines 70
cancel button in dialog 400, 403
caption See LCaption
Catch_ macro 281
chain of command 309
functions 309
maintaining 309
checking environment 287
ChooseDocument()
example 451
LDocApplication 439
class
interdependence in PowerPlant 97
name conventions 65
registering 288
class ID
reserved 138
when registering class 289
class library 75
ClearAttribute()
LWindow 358
click in a window 371
Click()
LPane 149
LWindow 372
ClickCell()
LTable 199
ClickInContent()
LWindow 372
ClickInDrag()
LWindow 371
ClickInGoAway
LWindow 371
ClickInGrow()
LWindow 368, 371
ClickInZoom()
LWindow 371
ClickMenuBar()
LEventDispatcher 323
ClickSelf()
LControl 225, 227
LListBox 155
LPane 142, 149
LWindow 367, 372
overriding 150
clipboard 543

clipping region 561
close
  dialog 408
  document 441, 443
  window 372
CloseBox window attribute 355
CloseDataFork()
  LFile 445
ClosePrinter()
  UPrintingMgr 493
CloseResourceFork()
  LFile 445
code library 74
coding conventions 65–70
command 307–315
  calling inherited commander 324
  chain of 309
  class hierarchy 308
  duty handling 312
  handling 314
  handling target 310
  identifying synthetic 323
  latency 313
  maintaining chain of command 309
  passing to supercommander 324
  responding to 323
  responding to quit as example 324
command hierarchy
  as design pattern 80
command number 318
  and menu item 115
  explained 316
  in Mcmd resource 319
  negative 319
  reserved numbers 318
  synthetic 319
  using synthetic 320
commander
  default 310
  defined 80
Compare()
  LComparator 557
CompareToKey()
  LComparator 557
constants name conventions 68
Constructor
  described 61
  to build controls 221
  to build dialogs 402
  to build panes 136
  to build printout 483
  to build views 180
  to build windows 362
constructor, stream 140
Contains()
  LPane 149
control 215–233
  as pane 217, 222
  broadcasting 230
  characteristics listed 218
  class hierarchy 215
  constructors 223
  creating 221–223
  creating on the fly 223
  defined 215
  deriving 224
  descriptor 219
  drawing 225
  hot spot in control 219
  installing in a view 223
  likely function overrides 224
  linking to listener 228, 232
  listening to other controls 224
  managing descriptor 226
  managing hot spot 226
  managing values 226
  message 218
  message components 220
  scroll bar 217
  setting port for 223
  standard Mac OS 217
  title 219
  using Constructor 221
  value 218
coordinate conversion 192
  and image size 193
coordinate systems 174–179
  global 175
  image 178
  listed 174
  local 177
  port 176
  window 176

CopyBits() (Mac OS)
and printing 491
CountPanels()
LPane 481
LPlaceHolder 479
LPrintout 477
LView 480
CreateNewDataFile()
LFile 445
CreateNewFile()
LFile 445
CreatePrintout()
LPrintout 483
CreatePrintRecord()
UPrintingMgr 493
CreateWindow()
LWindow 366
LWindow for dialogs 402
creator function 140, 288
and default commander 310
Current()
and variable length data 554
LArrayIterator 554
cursor, managing in a pane 149

D

Deactivate()
LPane 148
LWindow 371
UDesktop 376
UDesktop in dialogs 411
DeactivateSelf()
LPane 145, 148
LWindow 370
Debug_Signal 277
Debug_Throw 277
debugging 276–282
actions 278
Catch_ macro 281
errors and signals 277
macros listed 279, 280
options 277
signal side effects 279
strategy in PowerPlant 276
turning off 281
default button in dialog 400, 403
default commander 310
DelaySelect window attribute 357
DeleteAllSubPanes()
LView 188
deriving
application 274
controls 224
dialog 404
panes 140
printout 487
views 186
window 367
descriptor
in control 219
in document 442
in LSingleDoc 443
in pane 133, 147
design pattern defined 78
development process in PowerPlant 119
DevoteTimeToIdlers()
LPeriodical 508, 509
DevoteTimeToRepeaters()
LPeriodical 508
dialog 397–411
as listener 400
button tracking 400
characteristics 400
class hierarchy 399
closing 408
creating 402–406
creating on the fly 404
deriving 404
message handling 406, 407
mixing positive and negative messages 407
negative messages 406
overriding functions 404
pure modal in PowerPlant 399, 403
setting buttons 403
traditional 398
using Constructor 402
using Mac OS Dialog Manager 410
window kind 403
Dialog Manager (Mac OS) in PowerPlant 410
DialogSelect() (Mac OS) 398, 399
Disable()

LPane 148
disabled state 132
DisableSelf()
LPane 145, 148
DispatchEvent()
LApplication 290
DoAEClose()
LDocument 441
DoAEOpenOrPrintDoc()
LDocApplication 438, 488
DoAESave()
LDocument 441
tasks to implement 453
DoClose()
LDialogBox 408
LWindow 372, 373
document 434–443
and applications in design 435
closing 441, 443
constructor tasks 451
defined 435
design in PowerPlant 434
design responsibilities 435
implementing 451–454
in design pattern 87
list in application 440
opening 451
printing 489
resources in 454
reverting 454
saving 452
documentation for PowerPlant 59
DoDialog()
StDialogHandler 409
DontBeTarget()
LCommander 311, 314
DontRefresh()
LPane 144
DoPrint()
example 490
LDocument 441
tasks to implement 489
DoPrintJob()
LPrintout 476
DoQuit()
LApplication 273
DoRevert()
LDocument 441
DoSave()
LDocument 441
tasks to implement 453
DoSetBounds()
LWindow 368
DoSetZoom()
LWindow 368
drag and drop support 357
DragDeskWindow()
UDesktop 376
Draw()
LControl 225
LPane 143
LView 187
LWindow 142, 368
DrawCell()
LTable 199
drawing
and focus 187
control 225
on multiple monitors 562
pane 142
view 187
DrawSelf()
and printing 491
LControl 225
LPane 135, 142, 143
LPicture 199
LStdControl 143
LView 187
LWindow 143, 368
overriding 143
with multiple monitors 562
DrawWithJustification()
UTextDrawing 564
duty
and target 312
functions 314
handling 312
latency 313
dynamic menus 332

E

Enable()
LPane 148
enabled state 132
Enabled window attribute 356
EnableSelf()
LPane 145, 148
EndDrawing()
LGWorld 369
EndMarch()
UMarchingAnts 564
enumerated types name conventions 68
environment checking 287
EraseOnUpdate window attribute 356, 368
error See debugging
event dispatch
as design pattern 80
bottom-up 81
top-down 81
event handling in application 290
example code on CD 60
Execute()
LAttachment 517
ExecuteAttachments()
LAttachable 516
parameters 518
ExecuteSelf() 543
LAttachment 517
ExpandSubPane()
LView 188
ExportSelf() 543
LClipboard 544, 545

F

factored behavior in PowerPlant 101
factored classes in PowerPlant 99
factored design in PowerPlant 97
FailNIL_ macro 281
FailOSErr_ macro 281
FetchBottomFloater()
UDesktop 376
FetchBottomModal()
UDesktop 376
FetchIndexOf()
LArray 549
FetchItemAt()
LArray 549, 550, 551
LVariableArray 552
FetchMenu()
LMenuBar 332
FetchTopFloater()
UDesktop 376
FetchTopModal()
UDesktop 376
FetchTopRegular()
UDesktop 376
FetchWindowObject()
LWindow 377
file
in design pattern 87
name conventions 65
replacing existing 453
file I/O 434–454
FindCommandStatus()
and non-synthetic commands 328
and synthetic commands 330
in supercommander 329
LApplication 274
LCommander 315, 327
LDialogBox 405
LDocApplication 437
LDocument 440
LWindow 367
parameters 328
updates menu items 315
FindDeepSubPaneContaining()
LPane 150
LView 189
FindDominant Device()
UWindows 375
FindHotSpot()
LControl 227
FindNamedDocument()
LDocument 442
FindNamedWindow()
UWindows 375
FindNthWindow()
UWindows 375
FindPaneByID() 129, 139, 146, 172
LView 189

FindShallowSubPaneContaining()
LPane 150
LView 189
FindSubPaneHitBy()
LPane 149
LView 146, 189
FindWindowIndex()
UWindows 375
FinishCreate()
LControl 223
LPane 139
LView 186
FinishCreateSelf()
LControl 223
LDialogBox 401
LPane 139
LWindow 367
floating window 364
closing 374
focus 187
in application framework 84
setting 143
FocusDraw()
LView 143, 187
frame
binding 147
contents 135
for pane 129
LPrintout 474
framework
See also application framework
defined 75
difficulties using 76
from_End, iterating 554
from_Start, iterating 554
FrontWindowIsModal()
UDesktop 377
FSpExchangeFiles() (Mac OS) 454
FSRead() (Mac OS) 449
FSWrite() (Mac OS) 449

G

GetBytes()
LFileStream 449
GetClipboard() 543
LClipboard 544
GetCount()
LArray 549
GetCurrentMenuBar()
LMenuBar 332
GetCurrentPort()
UQDGlobals 377, 560
GetData()
LClipboard 543, 546
GetDataForkRefNum()
LFile 444
GetDataSelf() 543
LClipboard 544, 546
GetDefaultCommander()
LCommander 310
GetDefaultPrintRecord()
UPrintingMgr 493
GetDescriptor()
LControl 226
LDocument 442
LPane 142, 147
LSingleDoc 443
LWindow 360
GetDescriptorForPaneID()
LView 189
GetDocumentList()
LDocument 442
GetEOF() (Mac OS) 449
GetFPos() (Mac OS) 449
GetFrameBinding()
LPane 147
GetFrameLocation()
LPane 146
GetFrameSize()
LPane 146
GetImageLocation()
LView 189
GetImageSize()
LView 189
GetItemSize()
LArray 549
GetLastPaneClicked()
LPane 149
GetLatentSub()
LCommander 314

GetLength()
LStream 447
GetMacListH()
LListBox 154
GetMacMenuH()
LMenu 332
GetMacPort()
for windowKind 361
LPane 377
LWindow 377
GetMarker()
LStream 447
GetMaxValue()
LControl 226
GetMenuID()
LMenu 332
GetMinMaxSize()
LWindow 359
GetMinValue()
LControl 226
GetPaneID()
LPane 145
GetPrintJobSpecs()
LPrintout 476
GetQDGlobals()
UQDGlobals 560
GetResourceForkRefNum()
LFile 444
GetRevealedRect()
LView 190
GetScrollPosition()
LView 190
GetScrollUnit()
LView 190
GetSelectClick window attribute 357
GetSpecifier()
LFile 444
GetStandardSize()
LWindow 359
GetState()
LApplication 273
GetSubPanes()
LView 188
GetSuperView()
LPane 145
GetTarget()
LCommander 311
GetTopCommander()
LCommander 309, 310
GetUserCon()
LPane 147
GetValue()
LControl 226
LPane 142, 147
GetValueForPaneID()
LView 189
GetValueMessage()
LControl 226
GetWindowContent Rect()
UWindows 375
GetWindowStructure Rect()
UWindows 375
global coordinates 175
GlobalToPort Point()
LView 192
GrowZone() (Mac OS) 283

H

HandleAppleEvent()
LDocApplication 438
HandleClick()
LWindow 371
HandleCreateElementEvent()
LDocApplication 438
HandleKeyPress()
LCommander 315
LDialogBox 400
responds to keystroke 315
HasAttribute()
LPrintout 476
LWindow 358
HasFeature()
UEnvironment 288
hidden state 132
Hide()
LPane 148
LWindow 371
HideDeskWindow()
UDesktop 376
HideOnSuspend window attribute 356

HideSelf()
LWindow 370
hit testing in panes 149
HorizSBarAction()
LView 192
HorizScroll()
LView 192
host an attachment 512
hot spot 219
HotSpotAction()
LControl 225, 227
HotSpotResult()
LControl 225, 227
LStdCheckBox 240

I

idler periodical 507
image
as view characteristic 172
compared to frame 172
coordinates 178
managing in a view 189
size 193
ImagePointIsInFrame
LView 193
ImageRectIntersectsFrame()
LView 193
ImageToLocal Point()
LView 193
ImportSelf() 543
LClipboard 544, 545
inactive state 132
IncrementValue()
LControl 226
initialize
heap 282
QuickTime 283
Toolbox 283
Initialize()
LApplication 273, 274, 290
LApplication and menu creation 323
UQuickTime 283
InitializeHeap() 282
InitializeToolbox()
UQDGlobals 560
InsertCommand()
LMenu 332
InsertItemsAt()
LArray 549, 550, 551
LVariableArray 551, 552
Installing Resource Templates
Rez 577
InstallOccupant()
LPlaceHolder 479
integer types 67
InvalPortRect()
LPane 144
InvalPortRgn()
LPane 144
IsActive()
LPane 148
IsActiveScreenDevice()
UDrawingUtils 562
IsBroadcasting()
LBroadcaster 228, 230
IsColorGrafPort()
UDrawingUtils 562
IsDialogEvent() (Mac OS) 398, 399
IsEnabled()
LPane 148
IsEqualTo()
LComparator 557
IsEqualToKey()
LComparator 557
IsHitBy()
LPane 149
IsListening()
LListener 232
IsOnDuty()
LCommander 314
IsSyntheticCommand()
LCommander 320, 323
IsTarget()
LCommander 311
IsVisible()
LPane 148
ItemIsEnabled()
LMenu 333
iterator
example 555

iterator, multiple 555

K

key filter 153, 565
keystroke handling 314

L

LActiveScroller 173
LAddColumn()
    LListBox 154
LAddRow()
    LListBox 154
LApplication 104
    constructor 287
    See also application
LArray 97, 548–551
LArrayIterator 97, 553
latent commander 313
latent subcommander 106
LAttachable 93
    characteristics 516
    class hierarchy 513
    defined 512
    functions 516
LAttachable See also attachment
LAttachment 543
    class hierarchy 514
    data members 517
    defined 513
    deriving 520
    functions 517
    See also attachment
layers for windows 354
LBeepAttachment 522
LBorderAttachment 522
LBroadcaster 93, 97, 110
    and LControl 216
    defined 220
    described 227
LButton described 234
LCaption
    described 151
    descriptor in 133
    erasing text
LCicnButton
    described 235
LCLipboard 543–546
LClipboard
    GetData() 543
    SetData() 543
LCommandEnablerAttachment 523
LCommander 93, 105
    chain of command functions 309
    class hierarchy 107, 308
    command handling functions 315
    duty handling functions 314
    features 308
    See also command
    target functions 311
LComparator 557
LControl 93, 215
    and LBroadcaster 216
    class hierarchy 215
    See also control
LDataStream 449
LDefaultOutline described 151
LDialogBox
    class hierarchy 399
LDocApplication 104, 272
    class hierarchy 437
    command handling 437
    function interface 438
    See also application
LDocument
    class hierarchy 439
    command handling 440
    data members 440
    described 439
    functions 441
LEditField 93
    as periodical 510
    described 152
    descriptor in 133
LEraseAttachment 523
LEventDispatcher 105
LFile 111
    and data fork 446
    and resource fork 445
    as file systems 436
    class hierarchy 447

data members 444
described 444
functions 445
LFileStream 111
described 449
functions 449
LFocusBox described 152
LGrafPortView described 195
LGroupBox described 151
LGrowZone
as broadcaster 284
as periodical 285, 510
class hierarchy 284
described 284
using in non-PowerPlant projects 284
LGWorld 368
drawing in 369
LHandleStream 449
LIconPane described 152
LinkListenerToControls() 228, 229, 232
for dialogs 405
listen to message 232
listener
defined 86
in design pattern 86
linking to broadcaster 228, 232
ListenToMessage()
in memory management 284
LDialogBox 404, 405, 406, 407
LDialogBox sending commands 407
LListener 230, 232
LRadioGroup 232
lists
iterating 553
LKeyScrollAttachment 524
LListBox
compared to LTable 154
described 154
descriptor in 133
LListener 93, 97, 111
described 231
dialog as 400
LLockedArrayIterator 556
LLongComparator 558
LMenu 97
described 332
functions 332
using in non-PowerPlant code 333
LMenuBar 97
described 332
using in non-PowerPlant code 333
LModelObject 93
LMovieController
as periodical 510
described 152
LoadSystemTraits()
UTextTraits 570
LoadTextTraits()
UTextTraits 570
local coordinates 177
LocalToImage Point()
LView 193
LocalToPort Point()
LView 193
location, pane characteristic 129
LOffscreenView 368
described 195
LPaintAttachment 523
LPane 93, 109, 125
class hierarchy 126
descriptor in 134
printing functions 481
See also pane
value in 134
LPeriodical 93
class hierarchy 506
destructor 508
See also periodical
LPicture described 199
LPlaceHolder
characteristics 478
described 477
frame 482
functions 479
See also placeholder
LPreferencesFile 454
LPrintout
data members 475
frame 474, 482
See also printout

LRadioGroup 217
creating 245
LRunArray 552
LScroller 173
and controls 217
LSharable 559
LSingleDoc
data members 443
described 443
descriptor in 134
LStdButton described 240
LStdCheckBox described 240
LStdControl
described 239
descriptor in 134, 219
installing in a view 223
scroll bar 217
LStdPopupMenu
described 241
LStdRadioButton described 243
LStr255 558
LStream 111
class hierarchy 447
data members 447
described 447
functions 448
in document design 436
in stream constructor 141
overloaded operators 449
LString 97, 112, 558
LTabGroup described 153
LTable
compared to LListBox 154
described 197
LTextButton
described 236
descriptor in 134, 219
erasing text
LTextEditView
as periodical 510
described 196
LToggleButton described 235
LVariableArray 551
LView 109
class hierarchy 169
described 171
printing functions 480
See also view
LWindow 99
class hierarchy 352
described 351
descriptor in 134
See also window

M

MakeAlias()
LFile 445
MakeNewDocument()
example 450
LDocApplication 439
MakeScrollBars()
LScroller 217, 223
maximum size, window 358
MBAR resource 317
Mcmd resource 115, 317, 318
and command number 319
type 61
member access in PowerPlant 100
memory classes 285
memory management 283–287
allocating and deallocating 285
other strategies 286
reserve strategy 284
menu 316–333
adding dynamically 332
adding to application 322
clear a mark 329
command number See command number
default dispatch chain 323
effect of reorganizing 317
enable item 328
forcing update 327
mark an item 329
modify item text 329
modifying items 328
passing update to supercommander 329
real-time updating 326
removing dynamically 332
resources in PowerPlant 317
responding to choice 323
updating 325–331

updating in PowerPlant 326
updating negative items 330
updating non-synthetic items 328
updating synthetic items 330
when PowerPlant updates 327
when to update 325
working with Mac OS Menu Manager 332
MENU resource 317
MenuSelect() (Mac OS) 316, 320
message
broadcasting 230
components 220, 230
in control 218
in dialog 406
listening to 232
sending command as 220
messaging system design pattern 85
minimum size, window 358
mix-in class 92, 94
ModalDialog() (Mac OS) 398, 399, 409, 411
modeless dialog
window kind 404
mouse information in a pane 134
MouseEnter()
LPane 148
MouseLeave()
LPane 148
MouseWithin()
LPane 148
MoveBy()
LPane 146
MoveItem()
LArray 549
msg_AdjustCursor 149
msg_ControlClicked 230, 231
multiple inheritance
compared to single 95
in PowerPlant 92
mValue 220, 230
compared to mValueMessage 219
mValueMessage 218, 220, 230
compared to mValue 219

N

naming conventions 65–70
negative messages in dialogs 406
NewDeskWindow()
UDesktop 376
Next()
and variable length data 554
LArrayIterator 554
Normalize()
in drawing state classes 561
NormalizeWindowOrder()
UDesktop 377

O

ObeyCommand()
and command numbers 317, 323
and printing 489
and synthetic commands 321
calling inherited 324
LApplication 274, 324
LCommander 315
LDialogBox 404, 405, 407
LDocApplication 437
LDocument 440
LWindow 324, 367
responds to commands 315
typical functionality 323
occupant 478
installing in placeholder 479
removing from placeholder 479
offscreen drawing 368
OpenDataFork()
LFile 445
OpenDocument()
example 450
LDocApplication 438
opening a document 451
OpenOrCreateResourceFork()
LPreferencesFile 455
OpenPrinter()
UPrintingMgr 493
OpenResourceFork()
LFile 445
OrientSubPane()
LView 188
overloaded operators
defined 100

LStream 449
LString 559
overloading defined 100
overriding defined 100

P

page numbering 475
pane 125–150
adding to a view 187
characteristics 127–135
characteristics listed 128
characteristics of non-view 128
class hierarchy 126
class ID 138
constructors 139
contents 135
creating 136–140
creating on the fly 138
cursor management 149
defined 125
deriving 140
descriptor 133, 147
drawing 135, 142
frame 129
frame binding 147
hit testing 149
ID 129, 145
ID in Constructor 137
invalidating area 144
kinds of classes 109
likely function overrides 142
location 129
managing state 147
mouse information in 134
parts described 130
printing 491
removing from a view 187
size 129
state 131, 132
updating 144
using Constructor 136
value 133, 147
PaneIDT data type 129
panel
counting 481
in a pane 481
in printing 473
paper size 474, 482
parameter name conventions 66
pattern See design pattern
PeekData()
LStream 448
periodical 506–511
defined 506
flexibility 511
idlers 509
limitations 511
queues 507
removing from queue 508
repeaters 508
spending time 510
persistence 434
placeholder
alignment 478
in printing strategy 472
installing occupant 479
margins when printing 486
occupant 478
removing occupant 479
setting default alignment 486
PlaceInSuperFrameAt()
LPane 140, 146
PlaceInSuperImageAt()
LPane 140, 146
PointInHotSpot()
LControl 227
PointIsInFrame()
LControl 225, 227
port coordinates 176
port, getting Mac OS GrafPort 378
PortToGlobal Point()
LView 192
PortToLocal Point()
LView 192
PowerPlant
calling Toolbox routines 70
coding conventions 65–70
defined 19
documentation 59
example code on CD 60
factored behavior 101
factored classes 99

factored design 97
installing 58
installing resource templates 61–63
integer data types 67
member access 100
multiple inheritance in 92
resources 113
runtime requirements 58
source code folders 59
PP_Window_Kind described 360
PPob resource 61, 113
preferences file 454
Previous()
and variable length data 554
LArrayIterator 554
PrintCopiesOfPages()
LPrintout 476
overriding 487
PrintDocument()
LDocApplication 438
overriding 488
tasks to implement 488
printing 471–494
areas 482
creating a hierarchy 483–487
document 487–490
from the Finder 488
margins 486
page numbering 475
panel 473
panes 491
paper size 474, 482
view 491
window 489
printout
characteristics 485
creating on the fly 486
deriving 487
in printing strategy 472
overriding functions 487
using Constructor 483
PrintPanel()
LPane 481
LPrintout 477
LView 480
PrintPanelRange()
LPrintout 476
overriding 487
PrintPanelSelf()
LPane 481
LView 481
need to override 481
overriding 491
problem domain 75
ProcessCommand()
calling supercommander in dialog 406
in supercommander 324
LCommander 323
LDialogBox 405, 406
ProcessNextEvent()
LApplication 273, 290
profiling code 567
PutBytes()
LFileStream 449
PutChainOnDuty()
LCommander 314
PutInside()
LPane 140, 145
LView 187
PutOnDuty()
LCommander 314

Q

QuickDraw space 175
QuickTime, initializing 283

R

ReadAll()
LStream 448
ReadCString()
LStream 448
ReadData()
LStream 448
ReadDataFork()
LFile 445
LFileStream 450
ReadHandle()
LStream 448
ReadPString()
LStream 448
ReadPtr()

LStream 448
reconcile overhang 182
refCon
in Constructor 365
in Mac OS 365
in WIND 366
Refresh()
LPane 144
RegisterClass()
URegistrar 289
RegisterClass_() 288
registering PowerPlant classes 288
Remove()
LArray 549
RemoveAllAttachments()
LAttachable 516
RemoveAttachment()
LAttachable 516
RemoveCommand()
LMenu 332
RemoveItem()
LMenu 332
RemoveItemsAt()
LArray 549, 551
RemoveListener()
LBroadcaster 228
RemoveMenu()
LMenuBar 332
RemoveOccupant()
LPlaceHolder 479
RemoveSubCommander()
LCommander 309
repeater periodical 507
replacing files 453
ResEdit, resource templates for 62
ResetTo()
LArrayIterator 554
Resizable window attribute 355
ResizeFrameBy()
LPane 146
ResizeFrameTo()
LPane 146
ResizeImageBy()
LView 190
ResizeImageTo()
LView 190
Resorcerer, resource templates for 62
resource ID name conventions 69
resources
in PowerPlant 113
installing templates 61–63
reading and writing 445
Resume()
LWindow 370
UDesktop 376
reverting a document 454
Rez, resource templates for 62
RidL resource 61, 116, 229
customizing 229
Run()
LApplication 273, 290
LApplication and menu creation 322, 323
runtime requirements for PowerPlant apps 58

S

saving a document 452
SBooleanRect 131
scroll management 190
scrolling view ID 191
scrolling with an attachment 524
ScrollToPanel()
LPane 481
LPlaceHolder 479
LView 480
SDialogResponse structure 406
Select()
LWindow 370
SelectDeskWindow()
UDesktop 376
SendAECreateDocument()
LDocApplication 438
SendAEOpenDoc()
LDocApplication 438
SendAEQuit()
LApplication 273
SetAttribute()
LPrintout 476
LWindow 358
SetCancelButton()
LDialogBox 400

SetCommand()
LMenu 333
SetComparator()
LComparator 551
SetData()
LClipboard 543, 546
SetDataSelf() 543
LClipboard 544, 546
SetDefaultButton()
LDialogBox 400
SetDefaultCommander()
LCommander 310
SetDefaultView()
LView 186
SetDescriptor()
LControl 226
LPane 142, 147
LWindow 360
SetDescriptorForPaneID()
LView 189
SetEOF() (Mac OS) 449
SetFPos() (Mac OS) 449
SetFrameBinding()
LPane 147
SetHiliteModeOn()
UDrawingUtils 562
SetLatentSub()
LCommander 314
SetLength()
LStream 447
SetMarker()
LStream 447
SetMaxValue()
LControl 226
SetMinMaxSize()
LWindow 359
SetMinValue()
LControl 226
SetPaneID()
LPane 145
SetPortTextTraits()
UTextTraits 570
SetQDGlobals()
UQDGlobals 560
SetScrollUnit()
LView 190
SetSleepTime()
LApplication 273
SetSpecifier()
LFile 444
SetStandardSize()
LWindow 359
SetSuperCommander()
LCommander 309
SetTETextTraits()
UTextTraits 570
SetUpdateCommandStatus()
LCommander 327
SetUserCon()
LPane 147
SetValue()
LControl 226
LPane 142, 147
LStdRadioButton 231
SetValueForPaneID()
LView 189
SetValueMessage()
LControl 226
Show()
LPane 148
LWindow 371
ShowAboutBox()
LApplication 273, 274
ShowDeskWindow()
UDesktop 376
ShowNew window attribute 356
ShowSelf()
LWindow 370
signal macro
and side effects 279
signal macros 279
signal See debugging
SimulateHotSpotClick()
LControl 227
single-inheritance compared to multiple 95
size
for pane 129
window 358
SizeBox window attribute 356
source code folders 59

SPaneInfo 138, 140, 185, 186
SpendTime()
in memory management 285
LPeriodical 508
overriding 510
stack-based classes 112
memory 285
standard controls 217
standard state for window 359
StartBroadcasting()
LBroadcaster 228, 230
StartIdling()
LPeriodical 508
StartListening()
LListener 231
StartRepeating()
LPeriodical 508
StartUp()
LApplication 273, 274, 291
state
in a pane 131
in application 273
in windows 370
stationery files in PowerPlant 275
StClipRgnState 561
StColorPenState 561
StColorPortState 561
StColorState 561
StDeviceLoop 562
StDialogHandler 408
and periodicals 511
StEmptyVisRgn 561
StHidePen 561
StOffscreenGWorld 195
StopBroadcasting()
LBroadcaster 228, 230
StopIdling()
LPeriodical 508
StopListening()
LListener 231
StopRepeating()
LPeriodical 508
StPortOriginState 561
StProfileSection 112, 567
stream
defined 446
in design pattern 87
stream constructor 140
strings in PowerPlant 558
structure name conventions 69
StTextState 561
subpane defined 172
SuperPrintPanel()
LPane 481
LView 481
superview defined 128
Suspend()
LWindow 370
UDesktop 376
SViewInfo 185, 186
SwapItems()
LArray 549
SWindowInfo 366
for dialogs 404
SwitchTarget()
LCommander 311

T

TakeChainOffDuty()
LCommander 314
TakeOffDuty()
LCommander 314
target
and duty 312
becoming 311
defined 81
handling 310
switching 311
Targetable window attribute 357
TArray 552
TArrayIterator 556
text button See LTextButton
text traits 569
this->, use in PowerPlant 70
Throw_ macro 276, 277, 279
listed 280
TitleBar window attribute 355
TLockedArrayIterator 557
TrackHotSpot

LControl 225
TrackHotSpot()
LControl 227
TString 558
Txtr resource 61, 117, 570

U

UDebugging 98, 112
UDesktop
described 375
different versions 375
functions 376
used by LWindow 371
UDrawingState 560
UDrawingUtils 112, 562
UEnvironment 112
UKeyFilters 98, 112, 153, 565
UMarchingAnts 564
UModalDialogs 112, 410
UpdatePort()
LWindow 142
UProfiler 567
UResourceManager 568
UScreenPort 560
user state for window 359
UTextDrawing 564
UTextTraits 569
utilities
for windows 374
in an application framework 88
memory 285
utility classes 112
UWindows 112
described 374
functions 375

V

ValidatePrintRecord()
UPrintingMgr 493
ValidPortRect()
LPane 144
ValidPortRgn()
LPane 144
value
in a control 218
in a pane 133, 147
variable name conventions 66
VertSBarAction()
LView 192
VertScroll()
LView 192
view 169–194
add subpanes 187
as commander 85
characteristics 171–179
characteristics listed 171
class hierarchy 169
classes 109
constructors 185
coordinate systems 174
creating 180–186
creating on the fly 185
defined 169
deriving 186
drawing 187
image 172
image management 189
in application framework 83
in superview 181
pane information in 181
printing 491
reconcile overhang 182
remove subpanes 187
scroll management 190
subpanes 172
using Constructor 180
visible state 132
visual hierarchy
and controls 223
as design pattern 83
defined 170
finding top container 188
in application framework 83
in PowerPlant 110
maintaining 186, 187
managing panes in 145

W

window 351–378
as commander 352

as LWindow and WindowRecord 352
as pane 352
as view 352
attributes 353–358
attributes listed 354
characteristics 353–361
class hierarchy 352
click attributes 356
click in content 372
click in peripheral control 371
closing 372
creating 362–368
creating on the fly 366
deriving 367
descriptor 360
drawing 368
drawing attributes 356
drawing offscreen 368
floating 355, 364, 374
handling click 371
hide 370
in document design 436
layers 354
maximum size 358
minimum size 358
modal 355
overriding functions 367
pane information in 363
peripheral controls 355
positioning 365
printing 489
refCon 365, 366
regular 355
select 370
setting attributes at runtime 357
setting attributes in Constructor 364
setting window kind in Constructor 365
setting window kind in WIND 366
show 370
size 358
standard state 359
state 370
title 360
type 364
user state 359
userCon 365
using Constructor 362
utilities 374
window kind 360
Window Manager (Mac OS) 377
zooming 359
window coordinates 176
window kind
getting in PowerPlant 361
setting in Constructor 365
setting in WIND 366
use in PowerPlant 360
Window Manager (Mac OS) 377
WindowIsSelected()
UDesktop 376
WriteCString()
LStream 448
WriteData()
LStream 448
WriteDataFork()
LFile 445
LFileStream 450
WriteHandle()
LStream 448
WritePString()
LStream 448
WritePtr()
LStream 448

Z

Zoomable window attribute 356
zooming a window 359

CodeWarrior

The PowerPlant Book

Credits

authors: Carl B. Constantine, Jim Trudeau

editing Frank Turovich, the MOST Online PowerPlant class

example code Richard Atwell, Carl B. Constantine, John Daub, and Kenneth Drycksback

engineering: John Daub and Greg Dow

frontline warriors: the MOST Online PowerPlant class and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows